BuL
Virgo Gravitational Wave Searches Library for Burst Sources

User's Manual

v3r4

7th January 2005

VIR-MAN-LAL-7400-100

N. Arnaud
M.-A. Bizouard
F. Cavalier
A.-C. Clapson
M. Davier
G. M. Guidi
P. Hello
N. Leroy
T. Pradier
A. Viceré

Laboratoire de l'Accélérateur Linéaire
INFN & Università di Urbino

Contact : M.-A. Bizouard (mabizoua@lal.in2p3.fr)

Contents

Chapter 1

Introduction

1.1 Generalities

This library contains several packages dedicated to the search of burst gravitational waves (GW). It contains some facilities to generate the GW wave forms and place the templates, and the algorithms dedicated to look for GW. In addition to this burst specific library, BuL provides some applications related to the study of GW sources, such as spectral estimation, noise model generators, linear time domain filtering and statiscal algorithms.
This library is written in C++ and uses the C++ standard library and STL classes (complex, vector, string, ...). BuL uses also some external packages (for example the FFTW C library package (version 3.0.1)). It is developped on DEC/OSF1 V5.2, Linux/RH 7.2. It has been successfully compiled with gcc version 3.3.2. All the packages are managed and built using CMT.

The different packages are the following:

- BuM: contains some mathematical functions and some usefull C++ classes (a C++ class to embed the FFTW C functions for instance).
- BuS: contains some C++ classes to generate some determinist wave forms for Burst events and the Zwerger and Mueller waveforms catalogue.
- BuF: contains a C++ virtual class to code the different burst search algorithms. It contains also an example implementation of the BuD C++ virtual class, and all the different algorithms dedicated to Burst sources (ALF, PC, MF, NF).
- BuT: contains some C++ classes to place templates for Bursts search (1D and 2D templates placement).
- BuC: contains some C++ classes to perform Burst ITF coincidence search.
- Algorithm: contains some C++ virtual classes dedicated to all kind of algorithms except the time doamin linear filters contained in the Filter package.
- LinearFilter: contains C++ classes dedicated to discrte-time domain filters.
- Spectral: contains C++ classes dedicated to spectra estimation methods.
- Noise: contains some C++ classes to generate noise (white and colored noise).
- ITF: contains some header files which include all the characteristics of the dectectors.
- TeF: contains C++ classes dedicated to time frequency algorithms developed for the burst search

- Tiler: contains C++ classes providing a geometrical method to tile a 2D parameter space. It is used for Damped Sine and Gaussina Sine signals and has been tested for Coalescing Binaries
- BuLSet: logical package whose role is to manage easily the other BuL packages (see Installation section).

To use the BuL it is mandatory to use the namespace "_BuL" in all the user's applications.

1.2 BuL library installation

Since all the packages (including some of the external packages) are managed using CMT, it is necessary to install the CMT package first[1] if it is not already present.

BuL is making use of the external packages:

- FFTW (v3r0p1)
- DCPP (v2r1)

All the BuL packages can be downloaded from the LAL CVS repository or from the BuL Web site (http://www.lal.in2p3.fr/recherche/virgo/BuL/home.html). The architecture of this distribution is the following:

```
DataAnalysis/BuL/BuM           v3r5
DataAnalysis/BuL/BuS           v3r5
DataAnalysis/BuL/BuF           v4r4
DataAnalysis/BuL/BuT           v3r5
DataAnalysis/BuL/BuC           v1r3
DataAnalysis/BuL/ITF           v3r4
DataAnalysis/BuL/LinearFilter  v2r4
DataAnalysis/BuL/Noise         v3r5
DataAnalysis/BuL/Spectral      v2r4
DataAnalysis/BuL/Algorithm     v2r4
DataAnalysis/BuL/TeF           v1r8
DataAnalysis/BuL/BuLSet        v3r4
```

The installation of the BuL library is then easy following the instructions:

```
>cd BuL/BuLSet/vXrY/cmt
>cmt config
>cmt broadcast 'cmt config'
>cmt broadcast 'gmake'
```

The last command will send a broadcast command "gmake" to all the BuL packages in order to compile them.

1.3 C++ code style

- The use of pointers at the level of the users' program is source of troubles, errors and memory leakage problems. Moreover in most of the cases it is not needed, that is why, the BuL library defines and make use of objects in such a way that the user has never to manipulate pointers on objects. Those objects may be simple as basic containers or much more elaborate objects.
- The BuL library avoids as much as possible the copy of objects. Most of the time the classes methods API make use of the objects' address.

1.4 Namespace

The BuL library defines a namespace **_BuL** which embed all BuL classes and functions. It is then mandatory to declare the use of the _BuL namespace in any user's program.

Chapter 2

Bul packages description

In the following, the different classes and functions are described. The documentation is reduced to the public users' methods, in order to give a rapid overview of the present functionalities. For a complete reference, especially for the private fields and exact types reference, one will refer to the reference document provided by Doxygen[2]. On the contrary, the necessary *include file* names are systematically given.

2.1 Name convention

- f_0: sampling frequency
- $S(f)$: One-sided Power Spectral Density function also named PSD. The square root of S(f) is the One-sided Amplitude Spectral Density (ASD)

2.2 BuM: mathematics for bursts search

2.2.1 Class FFT

2.2.1.1 Fourier transform

The class FFT has been defined in order to provide a friendly interface to a lower level FFT algorithm library. This interface allows use of the complex container defined in the C++ DCPP package. This class is an interface to FFTW library[1].

We have chosen the following definition of the forward Fourier Transform (FT) of a function h(t):

$$\tilde{h}(f) = \int_{-\infty}^{+\infty} e^{-2\pi i f t} h(t) dt$$

The FFT class allows to perform one-dimensional Forward and Backward Discrete Fourier Transform (DFT) of real and complex discrete data sets.

The Forward DFT of an one-dimensional arry X_k of size N is given by

$$Y_k = \sum_{j=0}^{i=N-1} X_j e^{-\frac{2\Pi ijk}{N}} \tag{2.1}$$

The Backward DFT output is normalized such as the consecutive actions on a vector – $DFT_{Forward} \rightarrow DFT_{Backward}$ – provides the identical input vector.

2.2.1.2 FFTW description

The FFTW Fourier Transform provides an adaptative implementation of the Cooley and Tukey Fast Fourier Transform. To perform the FFT, FFTW creates a plan which is a data structure containing all the information that FFTW needs to compute the FT. The plan provides the best[2] combination of *codelets*[3] which computes FT of a given size. In other words, the size N of the FT is factorized such as the resulting recursive computation of smaller FT gives the fastest answer. This factorization is called WISDOM and can be saved on disk to be reused later on. This plan is created at runtime and is adapted to the hardware architecture. The definition of a plan depends basicaly on the size of the input data vector and on the FT direction. This particulary means that a plan can be (must be) reused as many times as identical FT have to be computed.

Different kinds of plan determination (wisdom) can be chosen:

- ESTIMATE: the factorization is guessed without measure. The result is immediate but not optimal.
- MEASURE: the executation time of many plans is measured with the pragmatic hypothesis that if an optimal plan for a size N is known, this plan is still optimal when size N is used for a larger transform. The drawback of this feature is that it can take a long time (from seconds to minutes depending on the machine and the precision of the clock timer) to create the plan, hence it is recommended only when one repeats FT request.
- PATIENT: improved algorithm compared to MEASURE especially adapted for long array but longer.
- EXHAUSTIVE: improved algorithm compared to PATIENT. This is the most optimal wisdom but the longest (minutes) and such should be used when a plan is used many times.

[1] FFTW 3.0.1 since release BuM/v4
[2] according to the execution time and not the number of floating point operations
[3] which are small piece of code generated at compile time

A *wisdom* can be saved on disk and then reused for any size which it is applicable as long as the planner option choice is not more "demanding" than this with which the wisdom has been cerated. For example a MEASURE wisdom can be used only for ESTIMATE and MEASURE but not for PATIENT and EXHAUSTIVE.

Finally, FFTW implements a DFT for complex data as well as a DFT for real data set. The latter gains roughly a factor 2 for the computation time and storage.

2.2.1.3 Class FFT description

The class FFT embeds the FFTW C library providing its main functionalities.

Five different constructors have been designed in order to fulfill different working cases.
As minimal information for the five constructors, one has to indicate:

- the size of the input vector
- the type of DFT: FFT_Real when the input vector is real, FFT_Complex when the input vector is complex. In case of real input data, the FFT_Real choice leads to a gain of factor 2 w.r.t. to FFT_Complex choice

When creating a FFT object, if the direction (FFT_Forward/FFT_Backward) is not specified, two plans are created: one corresponding to the Forward DFT and the other to the Backward DFT.

The other optional parameter conncerns the planner wisdom. If nothing is specified FFT is using the lower level algorithm (FFT_Estimate). The other alternative options are FFT_MEASURE, FFT_PATIENT and FFT_EXHAUSTIVE.
Once created, a FFT object is used to execute a Forward or a Backward DFT on a given vector of data. The type of data can be float or double, but the user has to take care of compiling his program with the corresponding version of the FFTW library (the binary directories are labelled by <tag>-float for a "float version" of FFTW and <tag> for the default "double version" of FFTW library. By default, all the BuL packages used the "double version" of FFTW.

The input of a complex data vector DFT (Forward and Backward) is stored in a complex data vector according to the following scheme:

- N is even
  - index 0: FT for f=0
  - index 1, ..., $\frac{N}{2}$: DFT output for $f = \frac{f_0}{N}$, ..., $\frac{N}{2}\frac{f_0}{N}$
  - index $\frac{N}{2} + 1$. ..., N-1 : DFT output for $f = -(\frac{N}{2} - 1)\frac{f_0}{N}$, ... $f = -\frac{f_0}{N}$
- N is odd: $n = \frac{N}{2}$ is the greater integer which is smaller than $\frac{N}{2}$
  - index 0: FT for f=0
  - index 1, ..., n: DFT output for $f = \frac{f_0}{N}$,..., $n\frac{f_0}{N}$
  - index $n + 1$. ..., N-1 : DFT output for $f = -n\frac{f_0}{N}$, ... $f = -\frac{f_0}{N}$

In the case of the Forward DFT of a real data vector the output can be either a complex data vector (whose format has been described above) or an half-complex data vector according to the following scheme[4]:

[4]One has to note that in the case of a real data vector DFT the use of the half-complex data format avoids a copy of the output vector although it is unavoidable when one requests a complex data vector

- N is even
  - index 0: FT for f=0
  - index 1, ..., $\frac{N}{2}$: real part of the FT for $f = \frac{f_0}{N}$, ..., $\frac{N}{2}\frac{f_0}{N}$
  - index $\frac{N}{2}+1$. ..., N-1 : imaginary part of the FT for $(\frac{N}{2}-1)\frac{f_0}{N}$, ... $f = \frac{f_0}{N}$
- N is odd: $n = \frac{N}{2}$ is the greater integer which is smaller than $\frac{N}{2}$
  - index 0: FT for f=0
  - index 1, ..., n: real part of the FT for $f = \frac{f_0}{N}$,..., $n\frac{f_0}{N}$
  - index $n+1$. ..., N-1 : imaginary part of the FT for $n\frac{f_0}{N}$, ... $f = \frac{f_0}{N}$

Include file "BuM_FFT.hxx"

Constructors
```
FFT (int aSize, FFT::Type aType)
FFT (int aSize, FFT::Type aType, FFT::Direction aDirection)
FFT (int aSize, FFT::Type aType, FFT::Wisdom aWisdom)
FFT (int aSize, FFT::Type aType, FFT::Direction aDirection, FFT::Wisdom aWisdom)
FFT (int aSize, FFT::Type aType, FFT::Wisdom aWisdom, char* aFileName)
FFT (int aSize, FFT::Type aType, FFT::Direction aDirection, FFT::Wisdom aWisdom,
            char* aFileName)
```

Public methods
```
int Forward (Vector<complex<double> >& input, Vector<complex<double> >& output)
int Backward (Vector<complex<double> >& input, Vector<complex<double> >& output)

int Forward (Vector<double>& input, Vector<complex<double> >& output)
int Backward (Vector<complex<double> >& input, Vector<double>& output)

int Forward (Vector<double>& input, Vector<double> & output)
int Backward (Vector<double> & input, Vector<double>& output)

void SetCheckVectorSize (bool aValue)
int ExportLastWisdomFile (char* fileName)
int SaveWisdomFile (char* fileName)
int GetSize ()
void SetDebug (bool debugValue)
bool GetDebug ()

int WriteFile (Vector<complex<double> > &aData, string aFileName, int aSampling,
            FFT::Output aPart)
int WriteFile (Vector<double> &aData, string aFileName, int aSampling)
int WriteFile (Vector<complex<double> > &aData, string aFileName, int aSampling,
            FFT::Output aPart, double aFMin, double aFMax)
int WriteFile (Vector<double> &aData, string aFileName, int aSampling, double aFMin,
            double aFMax)
```

In order to set up the FFTW options, one has defined several enumeration types (FFT::Type, FFT::Wisdom, FFT::Output and FFT::Direction). Those enumeration types are defined inside the class FFT. We give their definition below and in section 2.15.

Type	FFT_Complex	Output	Real
	FFT_Real		Imaginary
			Complex
			Modulus
Wisdom	FFT_Estimate	Direction	Forward
	FFT_Measure		Backward
	FFT_Patient		Both
	FFT_Exhaustive		

The *Forward(...)* method performs a forward FFT of a Vector of real (float or double) or complex (float or double) values. The output is stored in a complex Vector or in a real Vector (half complex format) depending on the choice of the user.

The *Backward(...)* method will perform an backward FFT of a Vector of complex values or of input Vcetor of half-complex values. The output is stored in a complex Vector.

When dealing with real Vectors, it is better to use the real FFT defined in FFTW (gain of a factor 2 in computional time) as shown in the following test programs example.

The method *SetCheckVectorSize(...)* allows to enable/disable the memory size of the input and output vectors check which is performed before any FT.

The *WriteFile(...)* methods prints out in a file the content of the output vector of a FT associated to the value of the corresponding frequency. In the case of a complex Vector output we can choose to print out the full complex representation, or the real, imaginary or modulus[5] part of the vector.

The *ExportLastWisdomFile(char* fileName)* method allows after creating a FFT object to save the last Wisdom into a file. Note that will correspond to the backward FT plan.

The *SaveWisdomFile(char* fileName)* method allows after creating a FFT object to save the wisdoms corresponding to forward and backward FT plan into 2 files. The file names are “Forward_fileName” and “Backward_fileName”.
Note that when creating a FFT object giving the fileName corresponding to the wisdoms, the constructors (IV & V) follow the file name syntax : “Forward_fileName” and “Backward_fileName’.

The method *GetSize()* gives the size of the plan hence the size of the data vector which are processed by the Fourier transform.

Program example (Complex FT):

```
int size = 1024;
Vector<double> input(size);
Vector<complex<double> > output(size);
Vector<complex<double> > end(size);

FFT A( size, FFT::FFT_Complex );

A.Forward( input, output );
A.Backward( output, end );
```

[5] the modulus of a complex number X is defined as $\sqrt{XX^*}$

Program example (Real FT):

```
int size = 1024;
Vector<double> input(size);
Vector<complex<double> > output(size);
Vector<double> end(size);

FFT B( size, FFT::FFT_Real );
B.Forward( input, output);
B.Backward( output, end) ;
```

2.2.2 Class Correl

This class provides several methods to perform special mathematical processing such as the correlation, the autocorrelation and the convolution of discrete data sets. All the methods are based on the Fourier Transform and hence make use of the FFT class. The main interest of the Correl object is due to the fact that the FFTW plans are determined only once when the object is created. The memory allocation of the temporary vector which are needed for each methods is also performed only once in the constructor. This feature will speed up the processing time in case of important call number to these spectral transforms.

Include file	"BuM_Correl.hxx"
Constructors	Correl(int aSize, bool aZeroPadding)

Public methods

```
int Correlation( Vector<double> &aDataOne, Vector<double> &aDataTwo,
                 Vector<double> &aData )
int Correlation( Vector<complex<double> > &aDataOne,
                  Vector<complex<double> > &aDataTwo,
                 Vector<complex<double> > &aData )
int AutoCorrelation(Vector<double> &aDataOne, Vector<double> &aData)
int AutoCorrelation(Vector<complex<double> > &aDataOne,
                 Vector<complex<double> > &aData)
int Convolution(Vector<double> &aDataOne, Vector<double> &aDataTwo,
                 Vector<double> &aData)
int Convolution(Vector<complex<double> > &aDataOne,
                 Vector<complex<double> > &aDataTwo,
                 Vector<complex<double> > &aData)

void SetDebug (bool aValue)
bool GetDebug ()
```

One can apply the different methods on zero padded data vectors or on normal data vectors.
In the case of zero padding, the dimension *aSize* which appears in the Correl constructor is the physical size of the data vector, and hence the input vectors size must be equal to $2 \times aSize$ (the input vector must contain the $aSize$ values followed by $aSize$ 0.) although the output vector size must also be equal to $aSize$.
In the case of no zero padding use, *aSize* is the dimension of the input and output vectors.

The correlation function of two functions g(t) and h(t) is defined as:

$$h * g(\tau) \equiv \int_{-\propto}^{+\propto} g(t+\tau)h(t)dt$$

The "correlation theorem" says:

$$\widetilde{h * g}(f) = \tilde{h}(f)\tilde{g}^*(f)$$

The convolution function of two functions g(t) and h(t) is defined as:

$$h \circ g(\tau) \equiv \int_{-\propto}^{+\propto} g(\tau-t)h(t)dt$$

The "convolution theorem" says:

$$\widetilde{h \circ g}(f) = \tilde{h}(f)\tilde{g}(f)$$

The practical implementation of the convolution and correlation functions are making use of the previously given properties.
The output vector is correctly ordering, taking into account the fact that the FFT algorithm provides a vector in a wrap around order.
In the case of zero padded vectors, the output vector is correctly formatted (spoiled data part removed).

Program example:

```
#include "math.h"
#include "BuM_Correl.hxx"

using namespace \_BuL;

int main( int narg, char* argc[] )
{
  int i;
  double x;
  double sigma = 1.;
  int size = 32;
  Vector<double> a(2*size);
  Vector<double> b(2*size);
  Vector<double> c(size);
  Vector<complex<double> > aa(2*size);
  Vector<complex<double> > bb(2*size);
  Vector<complex<double> > cc(size);

  for(i = 0; i < size; i++)
    {
      x = (i-size/2)*.5;
      a[i] = exp(-x*x/(2*sigma*sigma));
      b[i] = a[i];
      aa[i] = complex<double> (a[i], 0.);
      bb[i] = complex<double> (b[i], 0.);
      a[size+i] = 0.;  // zero padding
      b[size+i] = 0.;  // zero padding
      aa[size+i] = complex<double> (0., 0.);  // zero padding
      bb[size+i] = complex<double> (0., 0.);  // zero padding
    }

  cout << " Correl WITH ZERO PADDING " << endl;

  Correl toto(size, true);

  toto.Correlation(a, b, c);
  toto.Correlation(aa, bb, cc);

  toto.AutoCorrelation(a, c);
  toto.AutoCorrelation(aa, cc);

  toto.Convolution(a, b, c);
  toto.Convolution(aa, bb, cc);

  return( 0 );
}
```

2.2.3 Class GPSTime

This class provides some methods to convert a GPS time into UTC time or local system time (and vice versa). When converting a GPS Time into a UTC time one has to take into account the leap seconds which are regularly added to UTC time in order to ensure that the difference between a uniform time scale defined by atomic clocks (TAI time) does not differ from the Earth's rotational time by more than 0.9 second [4].
The UTC epoch is January 1, 1970.
The GPS epoch is January 6, 1980 (3600*24*(365*10+2+5)).

The first leap second added to UTC was introduced on June 30, 1972.
The last leap second added to UTC was introlduced on December 31, 1999.

```
As of January 1, 1999:
TAI is ahead of UTC by 32 seconds
TAI is ahead of GPS by 19 seconds
GPS is ahead of UTC by 13 seconds
```

All these information are stored in a file $BUMROOT/data/leapseconds.txt which is read when calling the GPSTime constructor.
To summarize, the GPS time conversion is since January 1, 1999 given by:

```
GPS = UTC - GPS_EPOCH + 13 seconds
```

Include file	"BuM_GPSTime"
Constructor	GPSTime
Public methods	time_t GPSToSys(time_t aGPSTime) time_t SysToGPS(time_t aSysTime) time_t GPSToUTC(time_t aGPSTime) time_t UTCToGPS(time_t aUTCTime) string TimeToString(time_t aTime) string GPSToUTCTimeString(time_t aGPSTime) string GPSToSysTimeString(time_t aGPSTime) string DateToString(time_t aUTCTime) string GPSToUTCDateString(time_t aGPSTime) string GPSToSysDateString(time_t aGPSTime) void SetDebug(bool aDebugValue) bool GetDebug()

2.2.4 Mathematical functions

Include file "BuM_Function.hxx"

Functions

```
void  Integral(Vector<double> aData, double aStep, double & aSum)
void  Integral(Vector<float> aData, float aStep, float & aSum)
void  IntegralNorm(Vector<double> aData, double aStep, double & aSum)
void  IntegralNorm(Vector<float> aData, float aStep, float & aSum)
void  IntegralModule(Vector<double> aData, double aStep, double & aSum)
void  IntegralModule(Vector<float> aData, float aStep, float & aSum)

double  GetThresholdKhi2(double aFalseAlarm, int aN )
double  GetThresholdGaussian(double aFalseAlarm, double aSigma )

int Correlation(Vector<double> &aDataOne, Vector<double> &aDataTwo,
                Vector<double> &aData)
int Correlation(Vector<complex<double> > &aDataOne,
                Vector<complex<double> > &aDataTwo,
                Vector<complex<double> > &aData)
int AutoCorrelation(Vector<double> &aDataOne, Vector<double> &aData)
int AutoCorrelation(Vector<complex<double> > &aDataOne,
                Vector<complex<double> > &aData)
int Convolution(Vector<double> &aDataOne, Vector<double> &aDataTwo,
                Vector<double> &aData)
int Convolution(Vector<complex<double> > &aDataOne,
                Vector<complex<double> > &aDataTwo,
                Vector<complex<double> > &aData)
```

Integral(...)	$\sum_{i=0}^{i=N} aData[i] * aStep$	N = aData.size()
IntegralNorm(...)	$\sum_{i=0}^{i=N} aData[i]^2 * aStep$	N = aData.size()
IntegralModule(...)	$\sum_{i=0}^{i=N} \|aData[i]\| * aStep$	N = aData.size()

GetThresholdKhi2(...)	returns the value of the threshold corresponding to the given false alarm rate *aFalseAlarm* asuming a χ^2 distribution of *aN* degrees of freedom.
GetThresholdGaussian(...)	returns the value of the threshold corresponding to the given false alarm rate *aFalseAlarm* asuming a gaussian distribution of width *aSigma.*

The folowing functions have been extracted from Numerical Recipes.

double Erfc(double aX)	erfc(x)
double Gammln(double aX)	$ln\Gamma(x)$
double Gammq(double aA, double aX)	Q(a,x) a>0
double Gammp(double aA, double aX)	P(a,x) a>0
void Gcf(double &aGammcf, double aA, double aX, double &aGln)	Q(a,x) a>0 x> a+1
void Gser(double &aGamser, double aA, double aX, double &aGln)	P(a,x) a>0 x> a+1

Note that the *erfc()* function is available in libc (math.h) on Linux RH 6.1. The erfc(...) function is defined as:

$$erfc(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$$

$$P(a,x) = \frac{\gamma(a,x)}{\Gamma(a)} = \frac{\int_0^x e^{-t}t^{a-1}dt}{\Gamma(a)}$$

$$\gamma(a,x) = \int_0^x e^{-t}t^{a-1}dt$$

$$\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$$

$$Q(a,x) = 1 - P(a,x) = \int_x^\infty \frac{e^{-t^2}t^{a-1}}{\Gamma(a)}dt$$

The functions *Correlation(...)*, *AutoCorrelation(...)*, and *Convolution(...)* are identical to the methods of the class **Correl** excepts that zero padding technics are not available.

2.2.5 Class Histogram

Include file	**BuM_Histogram.hxx**
Constructors	`Histogram (bool aSaveOverUnderFlow)`
Destructors	`~Histogram ()`

Public methods

```
int Define (int aBinNumber, double aMinVal, double aMaxVal,
            double aWeight)
int Define (double aBinSize, double aMinVal, double aMaxVal,
            double aWeight, double aScale)
int Add (double aVal)
double GetValue (int aBinIndex)
double GetMean ()
double GetSigma ()

int GetEntries ()
int GetBinNumber ()
double GetBinSize ()
int GetUnderFlowEntries ()
int GetOverFlowEntries ()
double GetWeight ()
double GetScale ()

int WriteFile (string aFileName)
int WriteHistogramFile (string aFileName)
int WriteBinHistogramFile (string aFileName)

void SetDebug (bool aValue)
bool GetDebug ()
```

This class provides a very basic implementation of an histogram object. At creation, one can switch on the option to save in a vector the value of the underflow and overflow entries.

The methods *Define (...)* defines the histogram either giving the bin number or the constant width of the histogram bins. The *aWeight* argument allows to give a specific weight for each entry (usually 1.). The *aScale* argument is used to rescale the value entered using the method *Add (...)*.

The method *Add (aVal)* adds en weighted entry in the bin corresponding to *aVal* (that means at bin i such as $aMinVal + i * aBinSize < aVal <= aMinVal + (i+1)aBinSize$).
The method *GetValue (aBinIndex)* returns the value of the bin number *aBinIndex.*

The method *GetMean ()* returns the mean of the histogram.

The method *GetSigma ()* returns the sigma of the histogram.

The method *GetEntries ()* returns the total number of entries.

The method *GetBinNumber ()* returns the number of bins.

The method *GetUnderflowNumber ()* returns the number of underflow entries.

The method *GetOverflowNumber ()* returns the number of overflow entries.

The method *GetWeight ()* returns the weight factor.

The method *GetScale ()* returns the scale factor.

The method *WriteFile ()* writes in a file the content of the histograms in a row wise format (main and overflow and underflow vectors if the switch has been selected).

The method *WriteHistogramFile ()* writes in a file the content of the histogram in a column wise format.

The method *WriteBinHistogramFile ()* writes in a file the bin value and its content of the histogram in a column wise format.
Program exemple:

```
#include "BuM_Histogram.hxx"

using namespace _BuL;

int main (int narg, char* argc[])
{
  int i;
  double temps;

  int taille = 1000;
  double frequency = 3000.0;
  double sigma = 2.0;

  Vector<double> signal (taille, 0.0);
  Histogram gaussienne (false);
  gaussienne.SetDebug (true);

  for (i = 0; i < taille; i++)
    {
      temps = ((double) (i - taille / 2)) / ((double) frequency);
      signal[i] = 100.0 * sin (2 * M_PI * (i - taille / 2) / frequency);
    }

  signal.WriteFile ("titi.dat");

  gaussienne.Define (100, 0, 100, 1.0);

  cout << "taille histo: " << gaussienne.GetBinNumber () << endl;

  cout << "remplissage histo" << endl;
  for (i = 0; i < taille; i++)
    gaussienne.Add (signal[i]);

  cout << "valeur a indice 30 : " << gaussienne.GetValue (30) << endl;
  cout << "valeur a indice 0 : " << gaussienne.GetValue (0) << endl;
  cout << "valeur a indice 100 : " << gaussienne.GetValue (100) << endl;
  gaussienne.WriteHistogramFile ("toto.dat");

  gaussienne.WriteFile ("tutu");

  cout << "mean: " << gaussienne.GetMean () << endl;
  cout << "sigma: " << gaussienne.GetSigma () << endl;

  return (0);
}
```

2.2.6 Class CumulativeHistogram

Include file	**BuM_CumulativeHistogram.hxx**
Constructors	`CumulativeHistogram ()`
Destructors	`~CumulativeHistogram ()`
Public methods	`int Define (const int aBinNumber, const double aMinVal, const double aMaxVal)`
	`int WriteSigmaHistogramFile (const string aFileName)`
	`int Add (const double aValue, const double aWeight)`

The class is derived from the *Histogram* class, to provide an alternative data accumulation. Instead of counting the number of events falling in a range of values, events are segregated following a first variable while a second one is saved so as to obtain its mean value and variance.
The *Define* method calls the *Hixstogram* class method, but does not require a weight value, as it is this weight that is modified for each event.
Add aggregates events according to the first variable, with the second variable stored to get mean and σ.
WriteSigmaHistogramFile returns the variance for each histogram bin, using the Histogram file format.
The mean values are obtained using *WriteHistogramFile* inherited from *Histogram*.

2.2.7 DataCard

The DataCard class may be used to read parameters stored in a file. Each line of the file starting with the keyword “@ “ will be considered. These lines contain a set of values associated to a keyword as shown in the following data card example. All other starting lines are seen as comments lines.

Include file	**BuM_DataCard.hxx**
Constructors	`DataCard (string aFileName)`
Destructors	`~DataCard ()`

Public methods	`int ReadCard ()`
	`int GetParam (string& aKey, int& aIndex, int& aValue)`
	`int GetParam (string& aKey, int& aIndex, float& aValue)`
	`int GetParam (string& aKey, int& aIndex, double& aValue)`
	`int GetParam (string& aKey, int& aIndex, string& aValue)`
	
	`void WriteCard ()`
	`void SetDebug (bool aValue)`
	`bool GetDebug ()`

The *ReadCard(...)* opens the file, reads and stored all keyword associated line in a list of strings. The method returns 0 except when a problem occured during the file open phase (return code = 1).

The *GetParam(...)* searchs for a value corresponding to the given position in the line associated to the given keyword. It is asumed that a keyword is unique.

In the following example one reads the example data card using the *GetParam(...)* methods.

```
# Example PARAMETERS DEFINITION
@ ALGO      Example
@ CLUSTER   Base
@ FILTERMEMORY No
#
@ NWINDOW   10
@ WINDOWS   10    20    30    40    50    100   150   200   250   300
@ THRESHOLD 31.26 30.84 30.66 30.70 30.42 29.98 29.58 28.68 28.54 28.20
```

Program example:

```
#include "BuM_DataCard.hxx"

using namespace _BuL;

int main( int narg, char* argc[] )
{
  int rc;
  int nWindow;
  double seuil;
  string name;

  double test;

  DataCard dc("../test/ConfigFile.txt");

  dc.SetDebug(true);
  dc.ReadCard();

  rc = dc.GetParam("NWINDOW", 0, nWindow);
  cout << nWindow << endl;

  rc = dc.GetParam("WINDOWS", 6, nWindow);
  cout << nWindow << endl;
  rc = dc.GetParam("WINDOWS", 10, nWindow);
  cout << nWindow << endl;
  rc = dc.GetParam("WINDOWS", 9, nWindow);
  cout << nWindow << endl;

  rc = dc.GetParam("THRESHOLD", 8, seuil);
  cout << seuil << endl;

  dc.GetParam("ALGO", 0, name);
  cout << name << endl;

  dc.GetParam("datasize", 0, test);
  dc.GetParam("TYTO", 0, test);


  cout << " write out dataCard" << endl;
  dc.WriteCard();

  return 0;
}
```

2.3 Spectral

This package contains several classes dedicated to non parametric spectral estimation algorithms. Most of the content of this package are well explained in [5].

The power spectral density function of a process $x(t)$ is defined as the Fourier Transform of the autocorrelation function $\mathcal{A}(\tau)$ of $x(t)$:

$$S(f) = \widetilde{\mathcal{A}(\tau)} \equiv PSD(f)$$

$$\mathcal{A}(\tau) = \int_{-\propto}^{+\propto} \mathrm{x(t+\tau)x(t)dt}$$

The amplitude spectral density function is defined as the square root of the power spectrum.

2.3.1 Non parametric spectral estimation

One usually tries to estimate the Power Spectral Density Function ($S(f)$) of a physical process using a data set composed of a given number N of discrete value $x_N(t)$. This induces a bias in all the PSD estimators which can be attributed to the existence of sidelobs in the Fejèr's kernel $\mathcal{F}(\{)$ which appears in the simple PSD estimator (periodogram):

$$< S_N(f) > = < \tilde{x}_N(f)\tilde{x}_N(-f) > \; = \int_{-\frac{1}{2}}^{\frac{1}{2}} \mathcal{F}(\mathrm{f'-f})\mathrm{S(f')df'}$$

where

$$\mathcal{F}(\mathrm{f}) = \mathrm{ND_N^2(f)} = \frac{\mathrm{sin^2 N\pi f}}{\mathrm{Nsin^2\pi f}}$$

The true PSD is convolved with the Fejèr's kernel (Fourier transform of the square function which is introduced by the size limitation of the data set). Note that the sidelobs amplitude of the Fejèr's kernel compared to the principal lob decreases when N is increasing. The sidelobs of the Fejèr's kernel introduce a leakage of power from one portion of the PSD to another. Indeed the sidelobs convolution involves frequencies which are distant from the one of $S(f)$. The estimated PSD is biased at frequencies for which $S(f)$ is small compared to the rest of the PSD. When $N \rightarrow \propto$ $\mathcal{F}(\mathrm{f-f'}) \rightarrow \delta(\mathrm{f-f'})$. $< S_N(f) >$ is hence an aymptotically unbiased estimator of $S(f)$.

In practise, one needs a mean to reduce the sidelobs effects which are especially critical for large dynamic range spectrum estimation. One technique, tapering[6], consists in multiplying in time domain, bin by bin, the discrete data set x_t by a window w_t function (data taper) and estimate the PSD of this new data set:

$$< S_N(f) >_{taper} = \int_{-f_0/2}^{f_0/2} \mathcal{H}(\mathrm{f'-f})\mathrm{S(f')df'}$$

where

$$\mathcal{H}(\mathrm{f}) = |\sum_{\mathrm{t=1}}^{\mathrm{N}} \mathrm{w_t e^{2\pi i f t}}|^2$$

The key idea behind tapering technique is to select the window function w_t such that $\mathcal{H}(\mathrm{f})$ has much smaller sidelobs than $\mathcal{F}(\mathrm{f})$. This leads to select taper which regularize the window at the extrema (see Figure 2.1). Nevertheless, this regularization induce an increase of the estimation variance due to the loss of data information at the extrema of the data analysis window. This drawback can be overcomed by performing an average estimation of the PSD.

[6]the other is pre-whitening which reduce the dynamic range

The most simple is to average the power spectrum using several data blocks or the same but with a reducing frequency resolution. The Welch approach [7] gives a better estimation: it consists of averaging overlapping data segments (WOSA approach). Indeed, with a overlap greater than 50 %, the data region which is underweighted at the extremity of one segment is taken into account in the next segment. One has to note that the overlapping feature recovers also some information concerning the autocorrelation contained in data values spanning two adjacent non overlapping data segments.

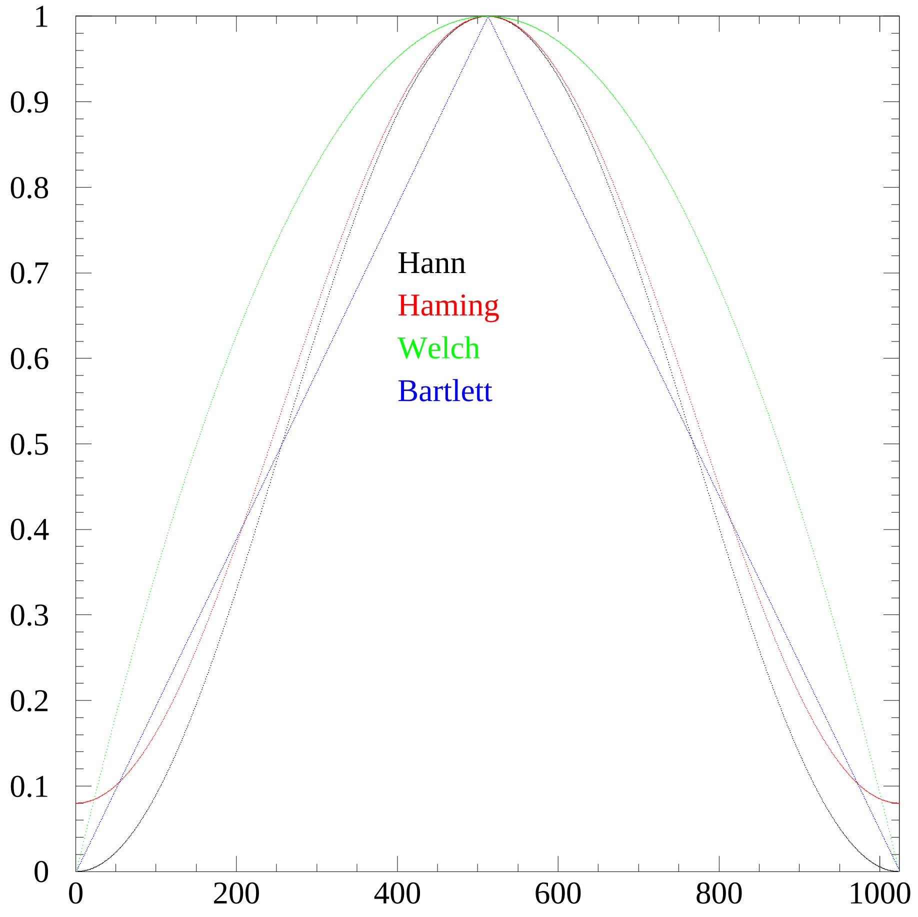

Figure 2.1:

Several window functions are usually used. For the choice of a particular function, there is a trade-off between having a principal lob as narrow as possible and the smallest sidelobs amplitude. BuL is providing the following tapers which are represented in Figure 2.1:

- Hann: $w_i = \frac{1}{2}(1 - cos\frac{2\pi i}{N-1})$
- Haming: $w_i = 0.54 - 0.46cos\frac{2\pi i}{N-1})$
- Welch: $w_i = 1 - (1 - \frac{2i}{N-1})^2$
- Bartlett $w_i = 1 - |\frac{2i}{N-1}|$

2.3.2 Class Taper

Include file	"Spectral_Taper.hxx"
Constructors	Taper()
	Taper(int aSize)

Public methods

int GetDataHann(Vector<double> &aWindow)
int GetDataHaming(Vector<double> &aWindow)
int GetDataWelch(Vector<double> &aWindow)
int GetDataBartlett(Vector<double> &aWindow)

int GetRawDataHann(Vector<double> &aWindow)
int GetRawDataHaming(Vector<double> &aWindow)
int GetRawDataWelch(Vector<double> &aWindow)
int GetRawDataBartlett(Vector<double> &aWindow)

void SetDebug(bool aDebugValue)
bool GetDebug()

The class Taper defines a enumeration type variable *Window* whose definition is given below and in section 2.15.

Window	None
	Hann
	Welch
	Haming
	Bartlett

The constructor *Taper()* creates the object without defining the size of the window. This means that the size must be assigned later during a object copy for example.
The other constructor *Taper(aSize)* creates the Taper object defining the size of the window.

The methods *GetDataHann(...), ...* fill the vector with the appropriate taper defintion. The window is normalized such as having $\sum w_j^2 = 1$.

The methods *GetRawDataHann(...), ...* fill the vector with the appropriate taper defintion. The window is not normalized.

Program exemple:

```
#include "math.h"
#include "Spectral_Taper.hxx"
using namespace \_BuL;

int main( int narg, char* argc[] )
{
  int i;
  int size = 128;
  Vector<double> a(size);
  Vector<double> b(size);

  Taper taping(size);
  taping.GetDataHann(a);
  taping.GetDataHaming(b);

  a.writeFile("hann.dat");
  b.writeFile("haming.dat");

  return( 0 );
}
```

2.3.3 Class Spectral

This class provides several methods to perform spectral estimation. All the methods are based on the Fourier Transform and hence make a use of the FFT class. The main interest of the Spectral object is due to the fact that the FFTW plans are determined only once when the object is created. The memory allocation of the temporary vector which are needed for each methods is also performed only once in the constructor. This feature will speed up the processing time in case of important call number to these spectral transforms.

Include file **Spectral_Spectral.hxx**

Constructors

```
Spectral (int aSegmentSize, Taper::Window aWindow, int aSegmentNumber,
            Spectral::PSDType aPSDType)
```

Destructors

```
~Spectral ()
```

Public methods

```
int Periodogram (Vector<double> &aData, Vector<double> &aPeriod)
int PSD (Vector<double> &aData, Vector<double> &aPSD, int aSampling)
int PSDWOSA (Vector<double> &aData, Vector<double> &aPSD, int aSampling)

int ASD (Vector<double> &aData, Vector<double> &aPSD, int aSampling)
int ASDWOSA (Vector<double> &aData, Vector<double> &aPSD, int aSampling)


int SpectrumPhase (Vector<double> &aData, Vector<double> &aPhase)

int SetWindow (Taper::Window aWindow, int aWindowMean)
int SetSegmentNumber (int aSegmentNumber)

int WriteSpectrumFile (Vector<double> &aData, string aFileName,
                        int aSampling)

int WriteSpectrumFile (Vector<double> &aData, string aFileName,
                        int aSampling, int aFMin, int aFMax)

double FlatenessOfPSD (Vector<double> &aPSD)
double LocalFlatenessOfPSD (Vector<double> &aPSD, int aSampling,
                            int aFMin, int aFMax)
double SigmaOfPSD (Vector<double> &aPSD, int aSampling)

bool GetEstimated ()

void SetDebug (bool aValue)
bool GetDebug ()
```

Several options are proposed to estimate a PSD:

- one-sided or two sided PSD
- with or without taper
- with or without averaging

- with or without Welch averaging approach (WOSA)

The one-sided and the two sided PSD are normalized such as (N = *aSegmentSize*):

- Two Sided
$$PSD(f) = \frac{< |FT(f)|^2 >}{f_0} \qquad f = 0, ..., \frac{f_0}{2}, -(\frac{N}{2} - 1)\frac{f_0}{N}, ..., -\frac{f_0}{N}$$

- One-sided
$$PSD(f) = 2 \times \frac{< |FT(f)|^2 >}{f_0} \qquad f = \frac{f_0}{N}, ..., \frac{f_0}{2}$$
$$PSD(0) = \frac{< |FT(0)|^2 >}{f_0} \qquad f = 0$$

  considering that $< |FT(f)|^2 >$ is the average over one or several FT data segments

Constructor arguments:

- *aSegmentSize*: size of the data segment which undergoes the Fourier Transform. The Two-sided PSD output vector size is equal to *aSegmentSize* although the one-sided PSD output vector size is *aSegmentSize/2*
- *aWindow*: taper window type (see class Taper for the list of available windows)
- *aSegmentNumber*: number of sub-segments of the input data vector which will undergo a FT and then be averaged. In the case of the PSD estimation with the Welch method the effective number of sub-segments is $2 \times aSegmentNumber - 1$
- *aPSDType*: one-sided (Spectral::OneSided) or two-sided (Spectral::TwoSided) PSD

Important remarq:
Be aware that the size of the data vector which is given as input of the different methods has to be equal to the product $aSegmentSize \times aSegmentNumber$.

Finally the power spectrum is normalized such that the result is in unit$/Hz$, and the amplitude spectrum is in unit$/\sqrt{Hz}$.

The *Periodogram(...)* method computes simply the square modulus of the FT of the data vector (no data windowing, no average, nor normalization are performed).

The *PSD(...)* and *ASD(...)* methods average the power spectrum over distinct data segments and make use of tapered data, while the *PSDWOSA(...)* and *ASDWOSA(...)* methods average the power spectrum over 50% overlapping data segments.

The *SpectrumPhase(...)* method computes the phase of the data vector. When the option "average" has been chosen in the constructor, the *SpectrumPhase(...)* method averages the FFT and then computes the phase.

The *WriteSpectrumFile(...)* methods write out in a file the spectrum (given in the arguments) associated to the corresponding frequency. One can also write out part of the spectrum corresponding to a given frequency band.

The *FlatnessOfPSD(...)* method computes the ξ parameter which estimates the spectral flatness of a PSD given as argument

$$\xi = \frac{e^{\frac{1}{N}\int_{-f_0/2}^{f_0/2} ln(PSD(f))df}}{\frac{1}{N}\int_{-f_0/2}^{f_0/2} PSD(f)df}$$

If PSD(f) is peaky $\xi \sim 0$.
If PSD(f) is flat $\xi \sim 1$.

The *SigmaOfPSD(...)* method computes the σ of the density spectral as following:

$$\begin{aligned} \sigma^2 &= \int_{-f_0/2}^{f_0/2} PSD(f)df \qquad \text{for a two} - \text{sided PSD} \\ \sigma^2 &= \int_{0}^{f_0/2} PSD(f)df \qquad \text{for a one} - \text{sided PSD} \end{aligned} \tag{2.2}$$

The *GetEstimated()* method returns true if a class to PSD, PSDWOSA, ASD or ASDWOSA has been already performed. Otherwise it returns false.

Program example:

```
#include "Spectral_Spectral.hxx"
using namespace \_BuL;
int main( int narg, char* argc[] )
{
  int i;
  double x;
  double sigma = 10.;
  int segment_size = 200000;
  int segment_number = 10;
  double khi;

  int size = segment_size*segment_number;

  Spectral titi(segment_size, Taper::Hann, segment_number, Spectral::TwoSided);

  Vector<double> aaa(size);
  Vector<double> bbb(segment_size);


  for(i = 0; i < size; i++)
    {
      x = (i-size/8)*.5;
      aaa[i] = exp(-x*x/(2*sigma*sigma));
    }

  titi.PSD(aaa, bbb, 20000);
  khi = titi.FlatnessOfPSD(bbb);
  cout << " flatness of PSD: " << khi << endl;

  titi.WriteSpectrumFile(bbb, "psd.dat", 20000);

  titi.PSDWOSA(aaa, bbb, 20000);
  khi = titi.FlatnessOfPSD(bbb);
  cout << " flatness of PSD WOSA: " << khi << endl;

  titi.WriteSpectrumFile(bbb, "wosa.dat", 20000);


  return( 0 );
}
```

2.4 Algorithm: algorithms for statistical analysis

2.4.1 Slicing window algorithms

A lot of algorithms are estimating a quantity over a chunk of data using a moving window (by step of one or several bins). Such algorithms need to keep in memory the value of the last data points in order to avoid discontinuities in the algorithm response at each new data chunk. The class **MovingSlice** has been designed to implement these facilities which are common to all the "slicing window algorithms". The other classes are derived from this base class.

2.4.1.1 Class MovingSlice

Let's assume one wants to compute a quantity with a sub-set of data corresponding to a window size ω and that one wants to repeat it moving the window by step of Δ bins.

parameter	definition	constraint
input data size	N	
window size	ω	$\omega \leq N$
slice step	Δ	$0 < \Delta \leq N$
output data size	$\frac{N}{\Delta}$	$\frac{N}{\Delta}$ is an integer
memory size	$\omega - \Delta$	

In the **MovingSlice** class one defines which part of the data chunk is used to compute the i^{th} element of the output vector of the algorithm. As shown in Figure 2.2:

- element 0: input data= the $\omega - \Delta$ last data of the previous data chunk + the Δ first data of the input data chunk
- element 1: input data= the $\omega - 2\Delta$ last data of the previous data chunk + the $2{*}\Delta$ first data of the input data chunk
- element $\frac{N}{\Delta} - 1$: input data= the ω last data of the data chunk

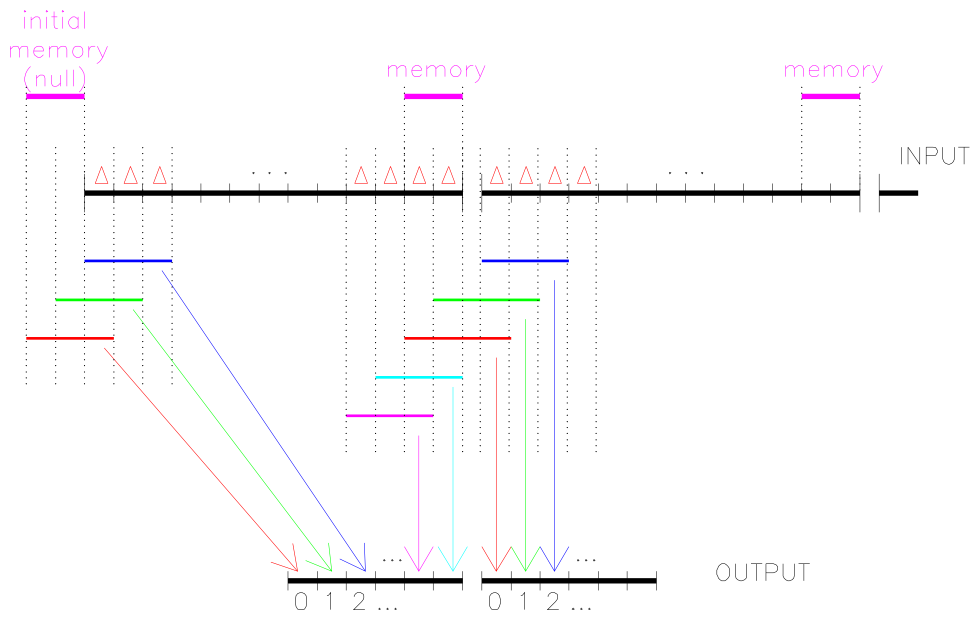

Figure 2.2: Slicing mechanism of the MovingSlice algorithm.

The content of the class **MovingSlice** is given for the sake of completion, but one can not instance such an object directly. It is used by other classes such as the Burst filters. There is only one public method. This content corresponds to the minimal content of any **Algorithm** class.

Include file	"Algorithm_MovingSlice.hxx"
Constructors	MovingSlice(int aInputSize, int aWindowSize, int aMovingDelta)
	MovingSlice(int aInputSize, int aWindowSize, int aMovingDelta, Vector<double> &aMemory)

Public methods

```
int SetMemory(Vector<double> &aMemory)
int SetMemory(double aValue)
void SetDebug(bool aDebugValue)
bool GetDebug()
```

Virtual method

```
int Compute(Vector<double> &aData, Vector<double> &aOut )
```

The *SetMemory(...)* methods allow to change the content of the memory of the algorithm at any moment.

2.4.1.2 Class Kurtosis

The Kurtosis estimator measures the degree of peakedness or the degree of fat tails of a distribution of probability density function $P(x)$. It is defined as:

$$\beta_2 = \frac{\mu_4}{\mu_2^2} \tag{2.3}$$

where μ_4 and μ_2 are the fourth and second central moments of the distribution of x which are defined as:

$$\mu_i =< (x- < x >)^i >= \int (x- < x >)^i P(x)dx \tag{2.4}$$

where $< x >$ denotes the expectation value.

The Kurtosis estimator for a Gaussian distribution is 3.

Include file	"Algorithm_Kurtosis.hxx"
Constructors	Kurtosis(int aInputSize, int aWindowSize, int aMovingDelta)

Public methods

```
void SetDebug(bool aDebugValue)
bool GetDebug()

int Compute(Vector<do4uble> &aData, Vector<double> &aKurto)
int ComputeIterative(Vector<do4uble> &aData, Vector<double> &aKurto)
int RecursiveEstimate(Vector<do4uble> &aData, Vector<double> &aKurto)
```

The *Compute(...)* method compute the Kurtosis quantity using the input Vector and fill the result in the output Vector. The Kurtosis estimation algorithm is not optimized since it assumes that the slice step (Δ parameter) may be different from 1 which implies no recursive relations between two estimations.

The *ComputeIterative(...)* method compute the Kurtosis quantity using the input Vector and fill the result in the output Vector. This Kurtosis estimation algorithm is more optimized than in the previous method since it assumes that the slice step (Δ parameter) is equal to 1 which implies some recursive relations between two estimations.

The *ComputeRecursive(...)* method compute the Kurtosis quantity using the input Vector and fill the result in the output Vector. This fast Kurtosis estimation algorithm implements the method described in [8].
This is not yet coded!

2.4.2 Mohanty's algorithm

2.5 LinearFilter: time domain linear filters

This package provides different classes for designing and applying digital filter in time domain

2.5.1 Introduction

We consider in this package linear digital filters which are linear and time invariant (LTI) system applied on discrete time sequence $x[n]$. A LTI system is entirely defined by its impulse response. That is the response to an impulse $\delta[n-k]$.

$$y[n] = T\{\sum_{k=-\infty}^{k=\infty} x[k]\delta[n-k]\} = \sum_{k=-\infty}^{k=\infty} x[k]T\{\delta[n-k]\} = \sum_{k=-\infty}^{k=\infty} x[k]h[n-k]$$

y[n] is then written as a convolution sum.

The LTI filters belong to two different categories:

- FIR (Finite Impulse Response): the impulse response has a finite number of non-zero values. This kind of filter is always stable
- IIR (Infinite Impulse Response): the impulse response has an infinite number of non-zero values. An IIR filter is stable if $h[n] = a^n u[n]$ with $|a| < 1$ and u[n] is the unit step discrete function.

It is usually easier to figure out the filter response in terms of zeros and poles representation instead of using the impulse response[7]. We consider in the following the sub-class of LTI such as:

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

We will consider (without loss of generality) that a_0 equals to 1.
Such a system is not absolutly causal, linear nor time invariant. Using the *z transfrom* the difference equation becomes:

$$\sum_{k=0}^{N} a_k z^{-k} Y[z] = \sum_{k=0}^{M} b_k z^{-k} X[z]$$

The *z transfrom*[8] of a sequence x[n] is the analog in discrete time domain of the *Laplace transform.*
z transfrom:

$$X(z) \sum_{k=-\infty}^{\infty} x[n]z^{-n}$$

Laplace transform:

$$\mathcal{L}(\mathrm{s}) \int_0^{\infty} \mathrm{x(t)e^{-st}dt}$$

The tranfer function of the system in the z plane is:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} b_k z^{-k}}{\sum_{k=0}^{N} a_k z^{-k}}$$

The transfer function can be factorized and one obtains a fractional equation:

$$H(z) = \frac{b_0}{a_0} \frac{\prod_{k=1}^{k=M}(1 - c_k z^{-1})}{\prod_{k=1}^{k=N}(1 - d_k z^{-1})}$$

[7]it should also be mentioned that the method of filter design by impulse invariance suffers from aliasing.
[8]bilateral definition

where c_k are the M complex zeros and d_k are the N complex poles of the tranfer function.

Note that the FIR case corresponds to a filter where all a_k are null except a_0 (H(z) has no pole except in z=0). An IIR filter has at least a pole ($\neq$0) which is not canceled by a zero.
The frequency response of the filter is given by the transfer function evaluated in the unit cercle. This is also the DFT of the impulse response.
Considering $z = e^{i\omega\Delta}$ the frequency response is:

$$H(e^{i\omega\Delta}) = \frac{b_0 \prod_{k=1}^{k=M}(1 - c_k e^{-i\omega\Delta})}{a_0 \prod_{k=1}^{k=N}(1 - d_k e^{-i\omega\Delta})}$$

where $\Delta = \frac{1}{f_s}$, c_k and d_k are complex variables.

The most general form o fthe transfer function of an IIR with real coefficients is:

$$H(z) = A\frac{\prod_{k=1}^{k=M_1}(1 - f_k z^{-1}) \prod_{k=1}^{k=M_2}(1 - g_k z^{-1})(1 - g_k^* z^{-1})}{\prod_{k=1}^{k=N_1}(1 - c_k z^{-1}) \prod_{k=1}^{k=N_2}(1 - d_k z^{-1})(1 - d_k^* z^{-1})}$$

where $M = M_1 + M_2$ and $N = N_1 + N_2$. The first-order factors represent real zeros at f_k and real poles at c_k. The seond-order factors represent the complex conjugate pairs of zeros and poles.
This expression suggests to implement an IIR filter as a cascade of first-order and second-order filters. One of the advantage is to provide a very general implementation to any order filter, provided the second-order filter implementation. The second advantage is to avoid numerical problem for high order filter design when using the bilinear transform. This filter structure has been chosen for the IIR class implementation.

The main issue in designing a digital IIR or FIR filter is to translate the **continuous-time** (analog) filter design into the **discrete-time** (digital) domain. That is to say to determine the a_i and b_i coefficients from poles and zeros given in the complex plane.

Let's start with the z domain transfer function

$$H(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2}}{1 + b_1 z^{-1} + b_2 z^{-2}} \tag{2.5}$$

There exists several methods to derive the coefficients. In the IIR class we have used the *bilinear transform* which maps region of s complex space to region of z complex plane: the left half s plane is mapping the interior of the z unit circle. The pure imaginary s axis is mapped onto the z unit circle.

$$\frac{s\Delta}{2} = \frac{1 - z^{-1}}{1 + z^{-1}}$$

$$z^{-1} = \frac{1 - \Delta s/2}{1 + \Delta s/2}$$

$$H(s) = \frac{A + Bs + Cs^2}{D + Es + Fs^2}$$

The biquadric filter coefficients are:

$$
\begin{aligned}
a_0 &= [\frac{C}{\Delta^2} + \frac{A}{4} + \frac{B}{2\Delta}]\frac{1}{x} \\
a_1 &= [\frac{A}{2} - 2\frac{C}{\Delta^2}]\frac{1}{x} \\
a_2 &= [\frac{C}{\Delta^2} + \frac{A}{4} - \frac{B}{2\Delta}]\frac{1}{x} \\
b_1 &= [\frac{D}{2} - 2\frac{F}{\Delta^2}]\frac{1}{x} \\
b_2 &= [x - \frac{E}{\Delta}]\frac{1}{x} \\
x &= \frac{1}{4}[D + 4\frac{F}{\Delta^2} + 2\frac{E}{\Delta}]
\end{aligned}
$$

Besides one defines second-order filter in s domain as follow:

$$1 + \frac{s}{\Omega Q} + \frac{s^2}{\Omega^2} \tag{2.6}$$

The roots are $s_\pm = \frac{\Omega}{2Q}[1 \mp \sqrt{1-4Q^2}]$. The main interest of this notation is that it ensures the stability of the filter since the real part of the poles are always negative.

We have adopted the following convention to describe a second-order and a first-order filter using equation (2.6).

- second-order: $\Omega \neq 0$ and $Q \neq 0$
  $1 + \frac{s}{\Omega Q} + \frac{s^2}{\Omega^2}$

$$
\begin{aligned}
A &= 1 \\
B &= \frac{1}{\Omega Q} \\
C &= \frac{1}{\Omega^2}
\end{aligned}
$$

- first-order: $\Omega \neq 0$ and $Q = 0$
  $1 + \frac{s}{\Omega}$: pole/zero at -Ω

$$
\begin{aligned}
A &= 1 \\
B &= \frac{1}{\Omega} \\
C &= 0
\end{aligned}
$$

- first-order: $\Omega = 0$
  s: pole/zero at 0

$$
\begin{aligned}
A &= 0 \\
B &= 1 \\
C &= 0
\end{aligned}
$$

- first-order: $\Omega = -1$
  1: pole/zero at ∞

$$A = 1$$
$$B = 0$$
$$C = 0$$

The last point which is worth mentioning concerning the bilinear transform technics is the frequency warping issue [9]. The relation between the **continuous-time** frequency Ω $(s = \sigma + i\Omega)$ and the discrete-time frequency ω associated to the z variable $(z = e^{i\omega})$ is:

$$s = \sigma + i\Omega = \frac{2}{\Delta}(\frac{1-e^{-i\omega}}{1+e^{-i\omega}}) = \frac{2i}{\Delta}tan(\frac{\omega}{2})$$

hence $\sigma = 0$ and $\Omega = \frac{2}{\Delta}tan(\frac{\omega}{2})$.

The bilinear transform is a method of squashing the infinite straigh analog frequency axis so that it becomes finite. But to avoid squashing the filter's desired frequency response too much the bilinear transform squashes the far ends of the frequency axis the most, leaving the middle part relativelty unsquashed as shown in Fig. 2.3.

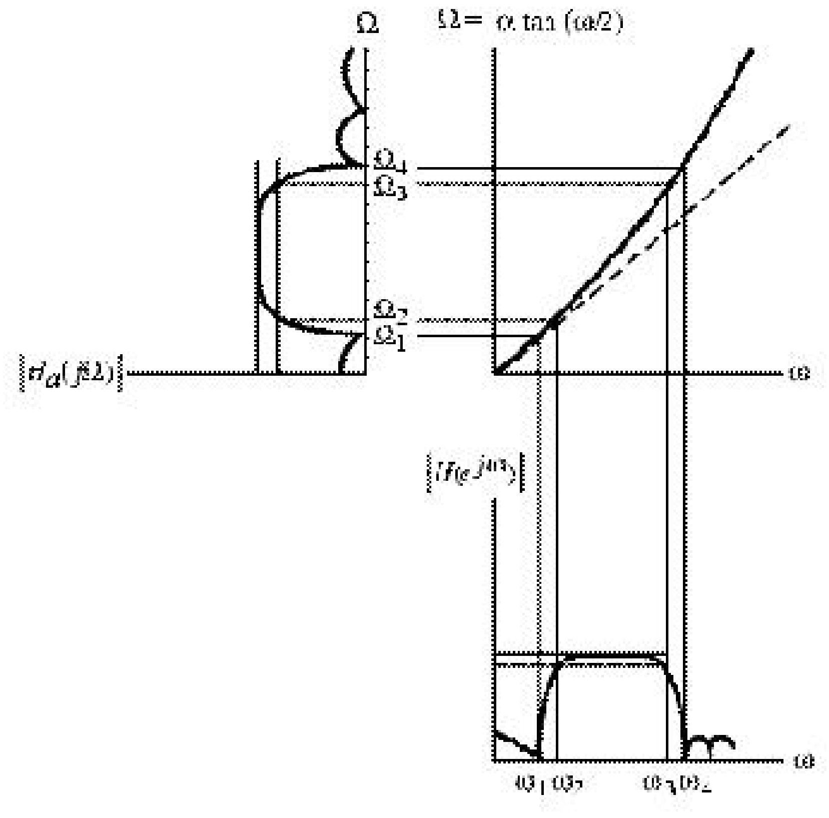

Figure 2.3: Frequency warping induced by the bilinear transfrom of a continuous-time pass band filter $H_c(j\Omega)$ into a discrete-time band pass filter $H(e^{j\omega})$. Extracted from [10].

This effect related to the non linear property of the bilinear transform changes the shape of the desired filter frequency response (especially in the transition bands region). It is acceptable when designing filters containing piecewise constant frequency magnitude characteristics such as High pass, Low Pass and Band Pass filters.

2.5.2 How to design a linear filter with class IIR using the zeros and poles design

The IIR class prevides an implementation of LTI digital filter providing the poles and zeros in continuous-time domain. It allows to define filters composed of numerous pass bands and attenuation bands whose caracteristics are constant. One has to specify the zeros and the poles of the filter following the convention given in section 2.5.1. We give in the following sections few examples of very simple desgined filters using IIR class.

2.5.2.1 Low pass filter

A low pass filter is realized using one pole with $\Omega = 100$ and Q=0 in s plane domain. The transfer function (see Fig. 2.4) is then

$$H(s) = \frac{1}{1 + s/\Omega}$$

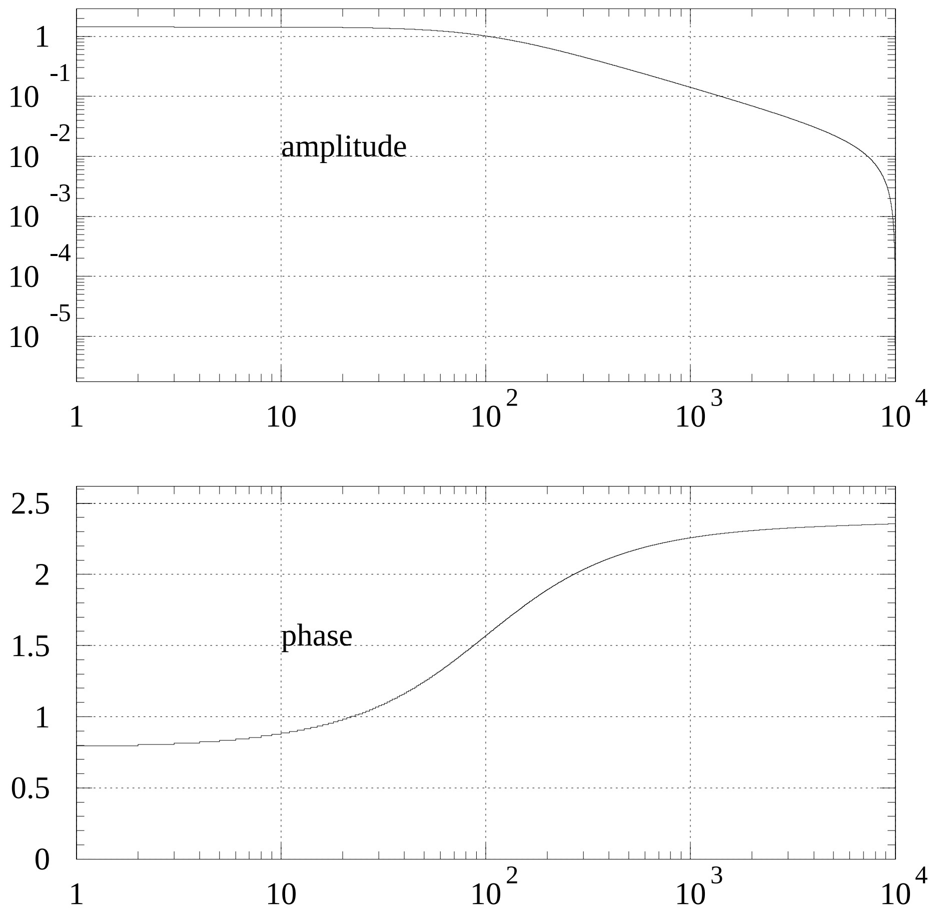

Figure 2.4: Transfer function (amplitude and phase) of a low pass filter designed using the IIR class. The x axis represents frequency in Hz.

2.5.2.2 High pass filter

The transfer function given in Fig. 2.5 has been obtained for and IIR filter containing:

- a triple poles at Ω=100 Hz
- a triple zero at Ω=0 Hz

The transfer function in s plane domain is

$$H(s) = (\frac{s}{1 + s/\Omega})^3$$

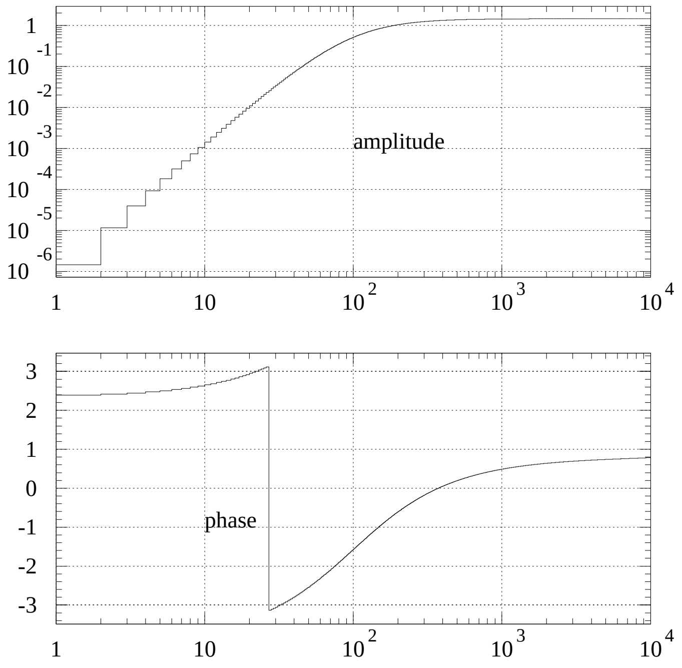

Figure 2.5: Transfer function (amplitude and phase) of high pass filter designed using the IIR class. The x axis represents frequency in Hz.

2.5.2.3 Multi poles and zeros filter

The transfer function given in Fig. 2.6 has been obtained for and IIR filter containing:

- 3 poles: (Ω, Q) = (10, 10), (100,10) and (1000, 1000)
- 1 zero: (Ω, Q) = (300, 10^5)

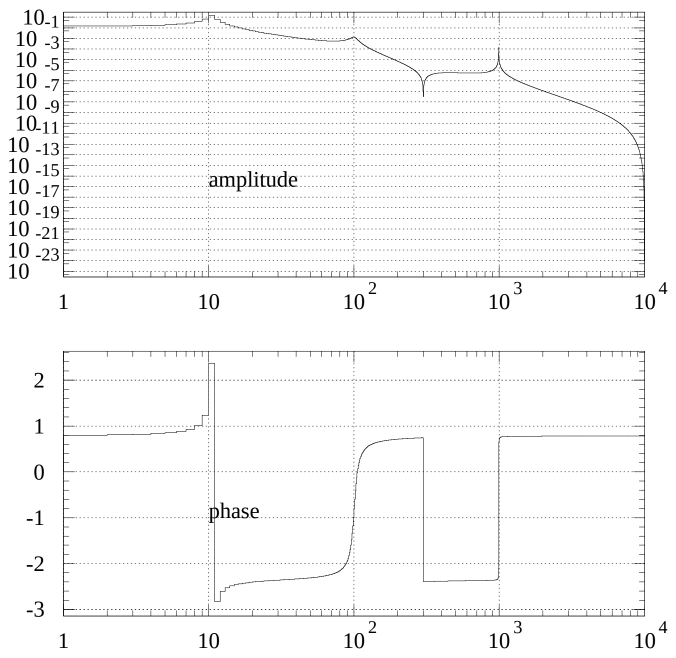

Figure 2.6: Transfer function (amplitude and phase) of a multi-pole and zero filter designed using the IIR class. The x axis represents frequency in Hz.

2.5.3 Class IIR

Include file **LinearFilter_IIR.hxx**

Constructors

```
IIR (Vector<double> &aZerosFrequency, Vector<double> &aZerosQ,
     Vector<double> &aPolesFrequency, Vector<double> &aPolesQ,
     double aGain, double aSampling)
```

Destructors

```
~IIR ()
```

Public methods

```
void SetGain (double aValue)
int SetGain (const double aValue , const int aRampTime)
double GetCurrentGain ()
void SetGainAtFrequency (double aValue, double aFrequency)
double GetGainAtFrequency (double aFrequency)

void ResetHistory ()
void SetHistory (int aDepth, Vector<double> &aX, Vector<double> &aY)
void GetHistory (int aDepth, Vector<double> &aX, Vector<double> &aY)

void PrintCoefficients ()

double Filter (double aInput)
int Filter (Vector<double> &aInput, Vector<double> &aOutput)

int GetTransferFunction (Vector<double> &aModulus,
                         Vector<double> &aPhase,
                         double aDeltaFrequency)
int GetTransferFunction (Vector<complex<double> >&aTransfer,
                         double aDeltaFrequency, bool aAlreadyInitialized)


int LowPassToHighPassTransform ()
```

The constructor specifies the set of poles and zeros that is the couple (Ω, Q) following the convention defined in section 2.5.1. One has also to give the global gain and the sampling frequency of the discrete time sequence to be filtered.

The *SetGain (...)* method allows to change the value of the global gain of the filter. It also possible to apply the change of gain with a ramp time given in sampling number.

The *GetCurrentGain ()* method returns the value of the global gain.

The *SetGainAtFrequency (...)* method allows to fix the gain of the filter at a given frequency.

The *GetGainAtFrequency (...)* method returns the gain at a given frequency.

The *ResetHistory ()* method allows to reset all the memory of the filter.

The *SetHistory (...)* method allows to set the history of the filter, that is the 3 last values of input and the 2 last values of filter output (not yet implemented).

The *GetHistory (...)* method returns the history of the filter (not yet implemented).

The *PrintCoefficients (...)* method print out the expression of all the second-order filters defined and used in cascade for implemented the requested filter as follow (multi-pole and zero filter described in section 2.5.2.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Filter composed by 3 filters of second order
Global Gain: 3
  --> Filter No: 0
 Filter in Laplace space
1 + 5.30516e-09 s + 2.81448e-07 s^2
-------------------
1 + 0.00159155 s + 0.000253303 s^2
  --> Filter No: 1
 Filter in Laplace space
1 + 0 s + 0 s^2
-------------------
1 + 0.000159155 s + 2.53303e-06 s^2
  --> Filter No: 2
 Filter in Laplace space
1 + 0 s + 0 s^2
-------------------
1 + 1.59155e-07 s + 2.53303e-08 s^2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

The *Filter (...)* methods perform the filtering transform either on a given input sequence value or on a full sequence vector. Note that in the case of a full sequence vector if one changes the filter gain it will be effective at the begining of the next sequence.

The *GetTransferFunction (...)* method computes the amplitude and the phase of the transfer function using Eq. 2.5. One has to specify the frequency step to fill out the vectors. there exists also the method which provides the complex transfer function.

The *LowPassToHighPassTransform ()* method transforms a low (high) pass filter into a low (high) pass filter as explained in 2.5.4.

Example: Low pass filter using IIR class

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fstream>
#include "DCPP_Vector.hxx"
#include "LinearFilter_IIR.hxx"
#include "Spectral_Taper.hxx"
#include "Spectral_Spectral.hxx"

using namespace _BuL;

// low pass filter
```

```
int main (int argc, char *argv[] )
{
  ofstream file;
  IIR *filter;

  double output;
  int i;

  double sampling = 20000;
  double gain;
  int datasize = 20000;

  Vector<double> module;
  Vector<double> phase;

  Vector<double> data (datasize);
  Spectral psd (datasize, Taper::Hann, 1, Spectral::OneSided);

  //  1 pole simple
  Vector<double> PoleFrequency (1);
  Vector<double> PoleQ (1);
  Vector<double> ZeroFrequency (0);
  Vector<double> ZeroQ (0);

  file.open ("filter.dat", ofstream::out);

  PoleFrequency[0] = 100.;
  PoleQ[0] = 0.;

  gain = 1.;

  filter = new IIR (ZeroFrequency, ZeroQ,
    PoleFrequency, PoleQ,
    gain, sampling);

  filter->PrintCoefficients ();

  filter->SetGainAtFrequency (1., 1.);

  cout << "Gain at 10 Hz: " <<  filter->GetGainAtFrequency (10) << endl;
  cout << "Global Gain: " << filter->GetCurrentGain () << endl;

  // Impulse reponse
  for (i = 0; i < datasize; i++)
    {
      if (i == 0)
        output = filter->Filter (1);
      else
        output = filter->Filter (0);
      file << (double)i/sampling << " " <<  output << endl;
      data[i] = output;
    }
  file.close ();
  psd.PSD (data, out, (int) sampling);
  psd.WriteSpectrumFile (out, "tf.dat", sampling);

  filter->GetTransferFunction (module, phase, 1.);
  module.WriteFile ("mod.dat");
  phase.WriteFile ("phase.dat");
  delete filter;
  return (0);
}
```

2.5.4 IIR filter using the Butterworth approximation

This class provides facilities to design low pass, high pass and band pass filters using the Butterworth approximation design [9]. The Butterworth filter provides the best Taylor Series approximation to the ideal lowpass filter response at analog frequencies. Butterworth low pass filters are defined by the property that the magnitude response is maximally flat in the pass band. For a N^{th} order low pass filter, this means that the firts (2N-1) derivatives of the magnitude squared function are zero at $\Omega = 0$. Another property is that the magnitude squared function is monotonic in the pass band and the stop band. The magnitude squared function for a continuous-time Butterworth low pass filter is of the form:

$$|H_c(j\Omega)|^2 = \frac{1}{1 + (\frac{j\Omega}{j\Omega_c})^{2N}} \tag{2.7}$$

Using $s = j\Omega$, Eq. 2.7 writes

$$H_c(s)H_c(-s) = \frac{1}{1 + (\frac{s}{j\Omega_c})^{2N}} \tag{2.8}$$

The roots of the denominator polynomial are therefore located at values of s:

$$s_k = j\Omega_c e^{(j\pi/2N)(2k+1)}, \qquad k = 0, 1, ...2N - 1 \tag{2.9}$$

There are hence 2N poles equally spaced in angle on a circle of radius Ω_c in the s plane. A pole never falls on the imaginary axis. On the contrary, if N is odd, poles exists at Ω_c and $-\Omega_c$. Finally note that poles always appear in complex conjugated pairs and that a poles at s_k corresponds a pole at $-s_k$. That implies that to build $H_c(s)$ one just needs to consider the N poles located on the left half plane part of the s plane (see Fig. 2.7). That insures the filter to be stable and causal.

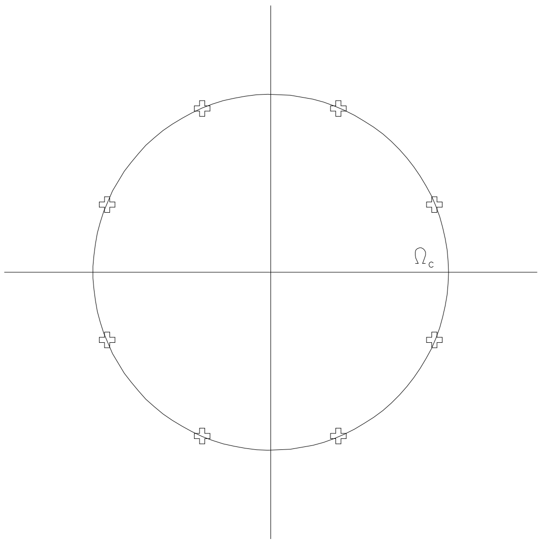

Figure 2.7: Location of the 8 poles of a Butterworth low pass filter of order N=4

The continuous-time transfer function writes:

- N is even

$$H_c(s) = \prod_{k=0}^{N/2} \frac{1}{(s - s_k)(s - s_k^*)} \tag{2.10}$$

- N is odd

$$H_c(s) = \prod_{k=0}^{N/2} \frac{1}{(s-s_k)(s-s_k^*)} \times \frac{1}{s+\Omega_c} \tag{2.11}$$

Grouping poles in complex conjugate pair the product factor if eq. 2.10 writes:

$$\frac{1}{s^2 - 2\Omega_c \mathcal{R}\mathrm{e}(\mathrm{s_k}) + \Omega_\mathrm{c}^2} \tag{2.12}$$

Once we have determined the poles, the low pass Butterworth filter can be implemented using the IIR class cascading second-order IIR filters defined by (Ω, Q). Using Eq. 2.12 it is easy to see that with reference of convention defined in section 2.5.1 one has:

$$\Omega_k = \Omega_c \tag{2.13}$$
$$Q_k = -\frac{1}{2cos(\pi(2k+1)/(2N)} \tag{2.14}$$

In case N is odd the single pole factor gives:

$$\Omega_k = \Omega_c \tag{2.15}$$
$$Q_k = 0 \tag{2.16}$$

Besides, a high pass filter can be constructed from a low pass filter by means of a transformation. Let's consider the low pass transfer function

$$H_l p(Z^{-1}) = \frac{a_0 + a_1 Z^{-1} + a_2 Z^{-2}}{1 + b_1 Z^{-1} + b_2 Z^{-2}} \tag{2.17}$$

To get the corresponding high pass filter let

$$Z^{-1} = -\frac{z^{-1}+\alpha}{1+\alpha z^{-1}}, \qquad \alpha = -\frac{cos(\frac{\omega_{lp}\Delta + \omega_{hp}\Delta}{2})}{cos(\frac{\omega_{lp}\Delta - \omega_{hp}\Delta}{2})} \tag{2.18}$$

Expanding Eq. 2.17 one gets for the high pass filter coefficients:

$$\begin{aligned} a_0 &= \tfrac{a_0 - a_1\alpha + a_2\alpha^2}{\delta]} & \delta &= 1 - b_1\alpha + b_2\alpha^2 \\ a_1 &= \tfrac{2\alpha a_0 - a_1(1+\alpha^2)+2a_2\alpha}{\delta} & b_1 &= \tfrac{2\alpha - b_1(1+\alpha^2)+2b_2\alpha}{\delta} \\ a_2 &= \tfrac{a_0\alpha^2 - a_1\alpha + a_2}{\delta} & b_2 &= \tfrac{\alpha^2 - b_1\alpha + b_2}{\delta} \end{aligned}$$

Choosing $\alpha = 0$ implies

$$\omega_{hp}\Delta + \omega_{lp}\Delta = \pi, \qquad \omega = 2\Pi f \tag{2.19}$$

and

$$\begin{aligned} a_0 &= a_0 & & \\ a_1 &= -a_1 & b_1 &= -b_1 \\ a_2 &= a_2 & b_2 &= b_2 \end{aligned}$$

Butterworth band pass filters are then built by cascading a high pass and a low pass filter on data. We give in the following section some example of low, high and pass band filters and their effect on a colored noise sequence.

2.5.4.1 Butterworth low pass filter

Low pass band frequency = 100 Hz. Order N=20.

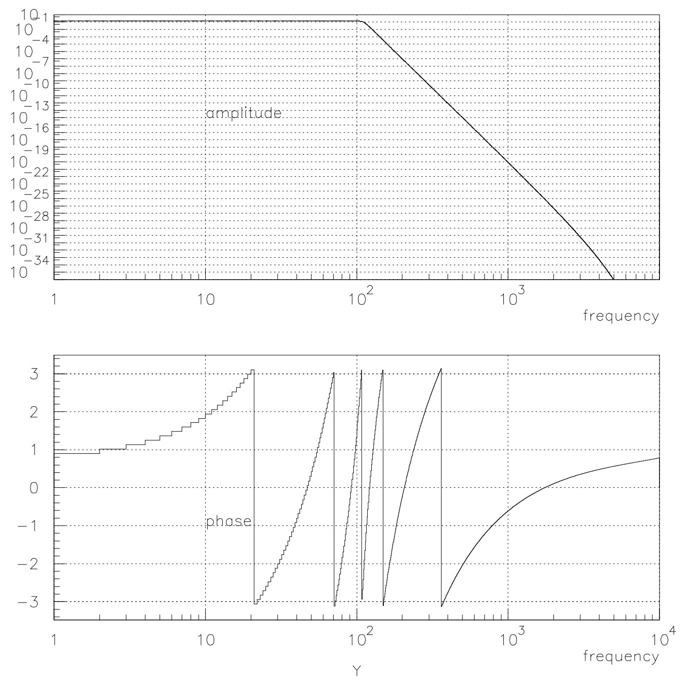

Figure 2.8: Amplitude and phase of the transfer function of a low pass band filter using the Butterworth approximation. N=20.

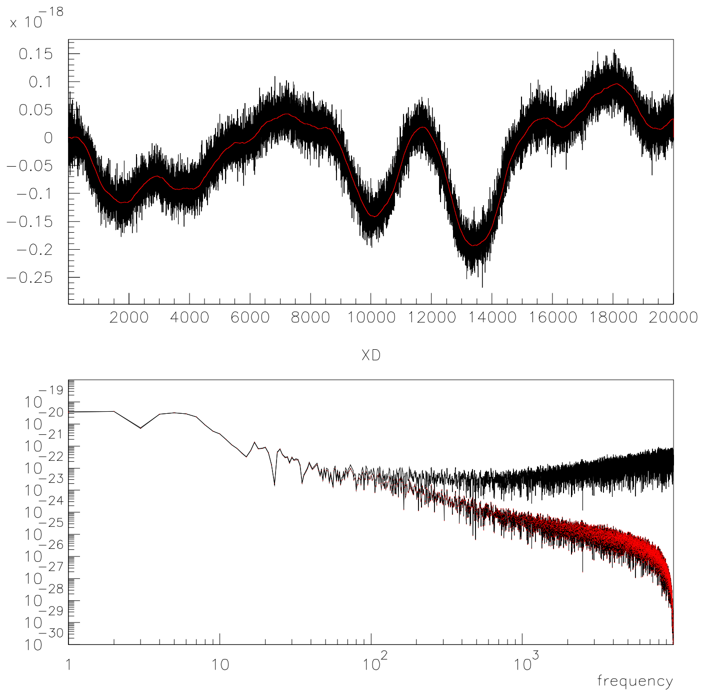

Figure 2.9: Data have been generated following the Virgo sensitivity curve (black curves) and are then low pass filtered. The red curves represent the filtered data (upper plot) whose PSD is represented below.

2.5.4.2 Butterworth high pass filter

order 30

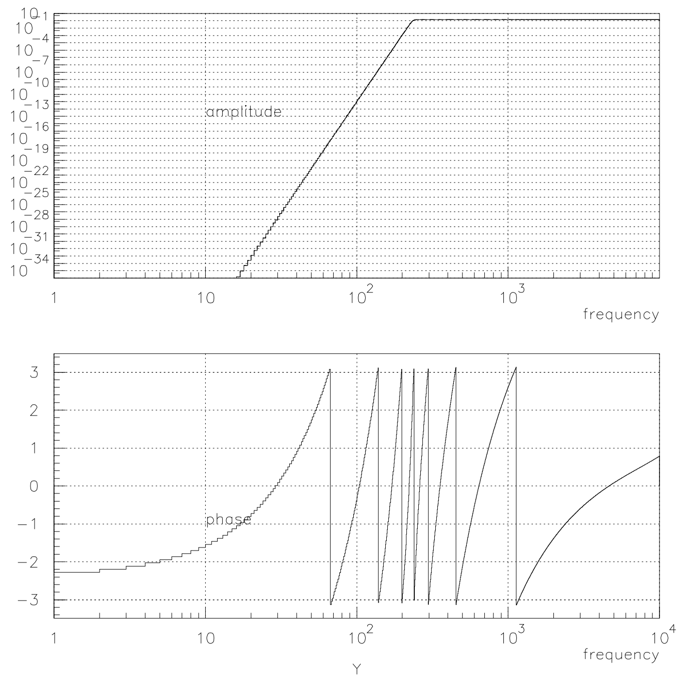

Figure 2.10: Amplitude and phase of the transfer function of a high pass band filter using the Butterworth approximation. N=30.

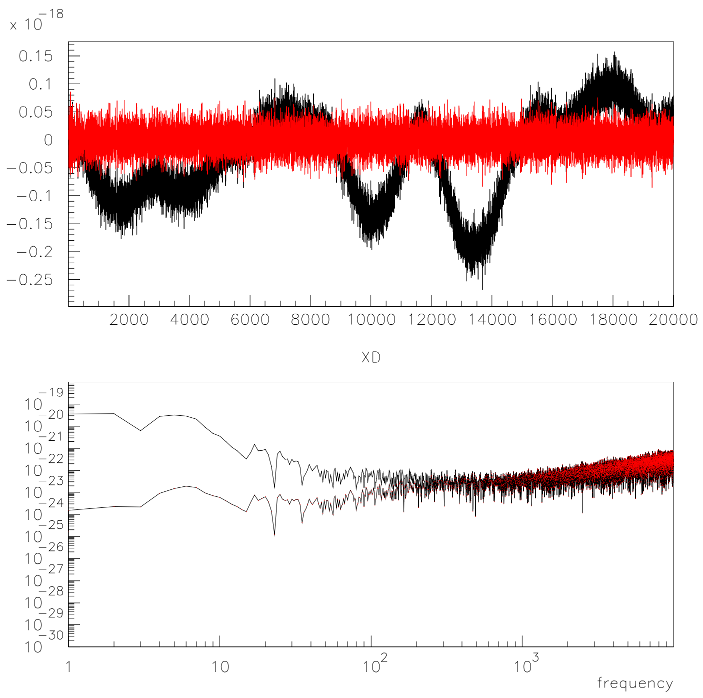

Figure 2.11: Data have been generated following the Virgo sensitivity curve (black curves) and are then high pass filtered. The red curves represent the filtered data (upper plot) whose PSD is represented below.

2.5.4.3 Butterworth pass band filter

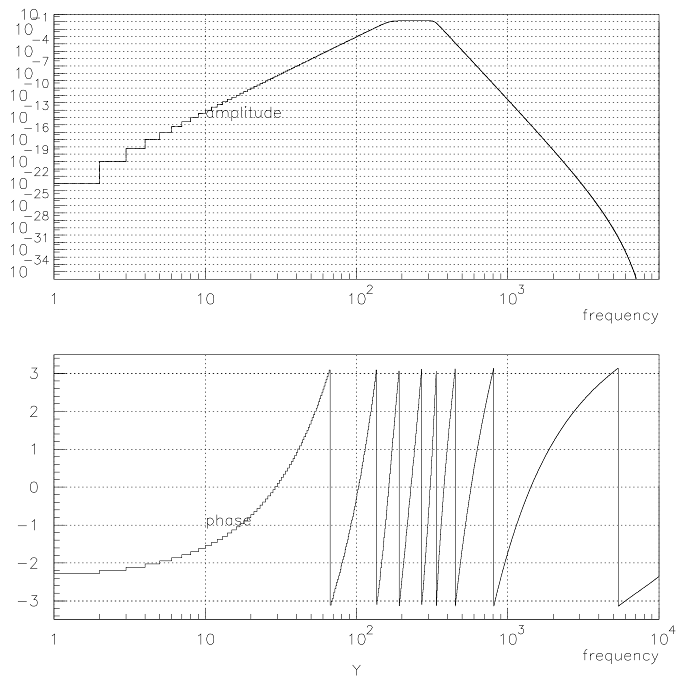

Figure 2.12: Amplitude and phase of the transfer function of a pass pass band filter using the Butterworth approximation. N=10 for the low pass filter and N=22 for the high pass filter.

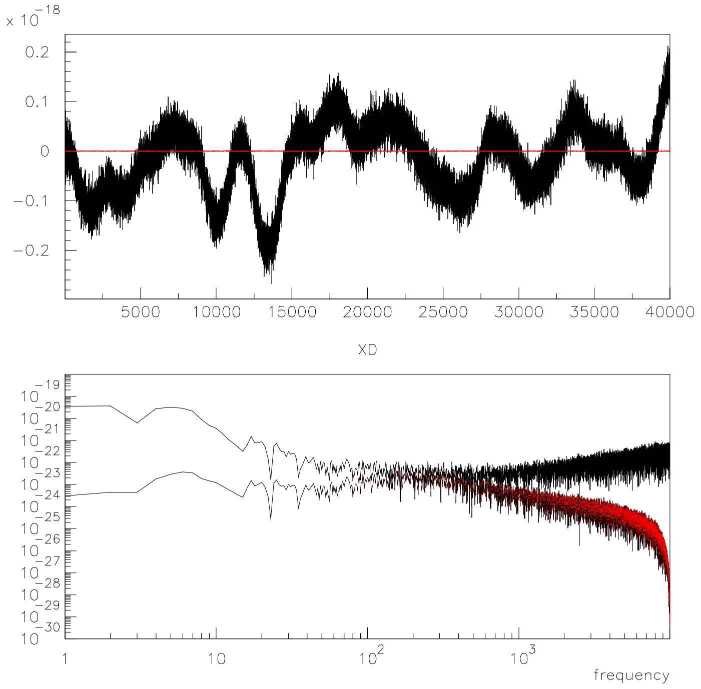

Figure 2.13: Data have been generated following the Virgo sensitivity curve (black curves) and are then band pass filtered. The red curves represent the filtered data (upper plot) whose PSD is represented below.

2.5.5 IIR filter using the Chebyshev approximation

2.5.6 Class Butteworth

This class implements 3 kind of filters:

- low pass filter
- high pass filter
- pass band filter

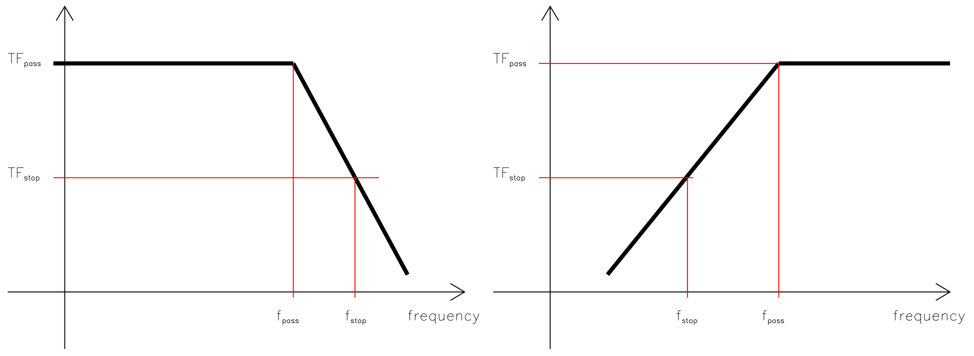

Figure 2.14: Transfer function of a low pass filter (left) and a high pass filter (right).

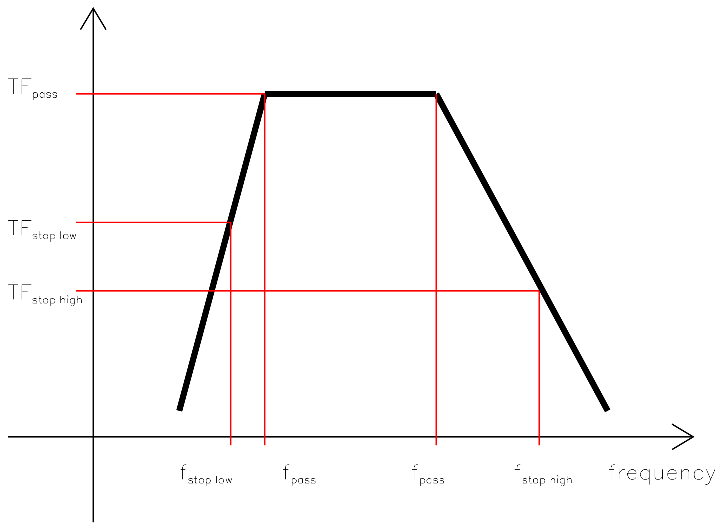

Figure 2.15: Transfer function of a pass band filter

A low pass and a high pass filter are defined by their stop band and pass band frequencies and the transfer function values corresponding to these frequencies (see Fig 2.14).

A pass band filter is defined by four frequencies and 3 transfer function values as given in Fig. 2.15.

Include file **LinearFilter_Butterworth.hxx**

Constructors

```
Butterworth (double aSampling, double aFrequencyPass,
             double aFrequencyStop, double aTFPass,double aTFStop)
Butterworth (double aSampling, double aFrequencyStopLow,
             double aFrequencyPassLow, double aFrequencyPassHigh,
             double aFrequencyStopHigh, double aTFStopLow,
             double aTFPassBand, double aTFStopHigh)
```

Destructors

```
~Butterworth ()
```

Public methods

```
double Filter (double aInput)
int Filter (Vector<double> &aInput, Vector<double> &aOutput)

int GetTransferFunction (Vector<double> &aModulus, e
            Vector<double> &aPhase, double aDeltaFrequency)
void PrintCoefficients ()

SetDebug (bool aValue)
bool GetDebug ()
```

The first constructor is used for defining a low and a high pass filter depending on the value of the stop band and pass band frequencies (see Fig. 2.14).
The second constructor is used to define a pass band filter. In all cases the filters consist to cascade second order butterworth low pass filter. The order of the resulting low pass filter is computed from the arguments given in the constructor.

The *Filter ()* method apply on a data input the time domain filter.

The *GetTransferFunction (...)* method computes the amplitude and the phase of the transfer function using Eq. 2.5. One has to specify the frequency step to fill out the vectors.

The *PrintCoefficients (...)* method print out the expression of all the second-order filters defined and used in cascade for implemented the requested filter as follow (multi-pole and zero filter described in section 2.5.2.

Example: Pass band filter using butterworth class

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fstream>
#include "DCPP_Vector.hxx"
#include "LinearFilter_Butterworth.hxx"
#include "Spectral_Taper.hxx"
#include "Spectral_Spectral.hxx"
```

```
#include "Noise_GaussianAR.cxx"

using namespace _BuL;

// Test program for pass band filtering

int main (int argc, char *argv[] )
{
  double tf_pass, tf_stop_low, tf_stop_high, frequency_pass_low, frequency_stop_low;
  double frequency_pass_high, frequency_stop_high;


  double sampling = 20000;
  int datasize = 20000;

  Vector<double> module;
  Vector<double> phase;

  Vector<double> data (datasize);
  Vector<double> output (datasize);
  Vector<double> out (datasize/2);


  tf_pass = .99;
  tf_stop_low = .01;
  tf_stop_high = .01;

  frequency_stop_low = 100;
  frequency_pass_low = 200;

  frequency_pass_high = 300;
  frequency_stop_high = 405;


  Spectral psd (datasize, Taper::Hann, 1, Spectral::OneSided);

  Butterworth filter (sampling, frequency_stop_low, frequency_pass_low,
      frequency_pass_high, frequency_stop_high,
      tf_stop_low, tf_pass, tf_stop_high);

  //  filter.PrintCoefficients ();


  filter.GetTransferFunction (module, phase, 1.);

  module.WriteFile ("mod.dat");
  phase.WriteFile ("phase.dat");

  // Gaussian colored noise
  char* Envvar;
  Envvar =  getenv ("NOISEROOT");
  string Ar_coef = Envvar;
  Ar_coef = Ar_coef + "/data/AR_Virgo_1024_4Hz.dat";
  GaussianAR gene (Ar_coef);


  gene.GetData (data);
  data.WriteFile ("data.dat");

  psd.ASD (data, out, (int) sampling);
  psd.WriteSpectrumFile (out, "spec.dat", sampling);

  filter.Filter (data, output);

  output.WriteFile ("filtered_data.dat");

  psd.ASD (output, out, (int) sampling);
  psd.WriteSpectrumFile (out, "spec_fil.dat", sampling);


  gene.GetData (data);
  data.AppendFile ("data.dat");
  filter.Filter (data, output);
  output.AppendFile ("filtered_data.dat");



  return (0);
}
```

2.5.7 Class MatchFilter

This class provide methods for performing the match filtering between a data vector and a given template.

Include file **LinearFilter_MatchFilter.hxx**

Constructors
```
MatchFilter (int aSampling, Vector<double> &aPSDTowSided,
             Vector<double> &aTemplate)
```

Destructors
```
~MatchFilter ()
```

Public methods
```
int SetPSD (Vector<double> &aPSDTowSided)
int SetTemplate (Vector<double> &aTemplate)
int SetTemplateAndPSD (Vector<double> &aTemplate, Vector<double> &aPSDTowSided)

int Filter (Vector<double> &aInput, Vector<double> &aOutput)
double GetNormalizationFactor ()

void SetDebug (bool aDebugValue)
bool GetDebug ()
```

When a *MatchFilter* object is created, the template is given and the normalization is done taking into account the two sided PSD. Note that the match filtering is performed using zero-padding technics. The *size* factor which is given corresponds to the size of the input data which will be filtered.

The method *SetPSD (...)* set the two sided PSD which is used for performing the match filtering. Note that the template is then re-normalized using the new PSD.

The method *SetTemplate (...)* set the template to the content given as argument. Note that the template will be normalized using the current PSD (given at object creation or using the *SetPSD ()* method. The vector containing the template must have a size equal or smaller than the size given in the constructor.

The method *Filter(...)* performs the match filtering between the input data vector and the template. The result is stored in the output vector.

The method *GetNormalizationFactor ()* returns the factor used for the template normalization.

Program example:

```
#include "math.h"
#include "LinearFilter_MatchFilter.hxx"

using namespace _BuL;

int main( int narg, char* argc[] )
{
  int i;
  double x;
  double sigma = 10.;
  int size = 10000;
  int sampling = 20000;

  Vector<double> patron (size);
  Vector<double> data (size);
  Vector<double> output (size);
  Vector<double> psd_twosided (2*size);
  double factor;

  double norm = sqrt (sqrt (M_PI) * sigma);

  for(i = 0; i < size; i++)
    {
      x = (i-size/2);
      patron[i] = exp (-x*x/(2*sigma*sigma))/norm;
    }

  psd_twosided.ReadFile ("../data/psd.dat");

  MatchFilter filter (sampling, psd_twosided, patron);
  filter.SetDebug (true);

  factor = filter.GetNormalizationFactor ();

  for(i = 0; i < size; i++)
    patron[i] = patron[i]/factor;

  // just to check the normalization
  filter.SetTemplate (patron);

  // just to check the PSD setting
  filter.SetPSD (psd_twosided);

  // autocorrelation
  data = patron;

  filter.Filter (data, output);

  output.WriteFile ("match.dat");

  return (0);
}
```

2.5.8 Class Goertzel

The Goertzel algorithm computes the DFT coefficient associated to the desired frequency, without the frequency sampling restriction inherent to the DFT: $\omega = 2\pi k f_0/N$.
The computation is propertionel to N^2 (let's recall that FFT algorithm computation goes as $NlogN$). The main advantage of the Goertzel algorithm is to restrict the computation to the desired frequencies. Given an input data size N, that still constraints the frequency resolution, the sampling frequency f_0 and the frequency of interest f_i, the algorithm computes the DFT coefficient at index $k = Nf_i/f_0$.

The Goertzel algorithm can be viewed as the response of a system to a finite length input x[n] sequence [9]. That's the reason why the algorithm is commonly implemented with a second-order IIR bandpass filter, with the following transfer function:

$$H_k(z) = \frac{1 - e^{\frac{j2\pi k}{N}} z^{-1}}{1 - 2\cos(\frac{2\pi k}{N}) z^{-1} + z^{-2}} \tag{2.20}$$

In the time domain, the DFT k^{th} coefficient X[k] is given by:

$$X[k] = s_k[N] + e^{-j2\pi k/N} s_k[N-1] \qquad \text{the sought DFT coefficient} \tag{2.21}$$

$$s_k[n] = x[n] + 2\cos\frac{2\pi k}{N} s_k[n-1] - s_k[n-2] \qquad \text{recursion relation} \tag{2.22}$$

$$s_k[-2] = 0 \qquad \text{initial conditions} \tag{2.23}$$

$$s_k[-1] = 0 \tag{2.24}$$

Include file **LinearFilter_Goertzel.hxx**

Constructors
```
Goertzel (bool aDebug)
```

Destructors
```
~Goertzel ()
```

Public methods
```
int Setup (double aSamplingFrequency, double& aLineFrequency,
           int& aChunkSize)

double Filter (Vector<double>& aData)
int Filter (Vector<double>& aData, Vector<double>& aLine)
int Filter (Vector<double>& aData, Vector<double>& aLine,
            Vector<double>& aWindow)

void Reset ()
```

The method *Setup ()* requires the values for f_0, f_i and N ($aChunkSize$). Once called the filter memory is initilized to 0.

Filtering either produces one DFT coefficient, for input *Vector* of size larger than N, or a *Vector* of values for the coefficient, computed on non-overlapping chunks of length N.

It is also possible to apply a window on the input data. The use of these additional operations is not very clear, since thefilter deals with one frequency only, at a resolution fixed by the size of the non-null part of the data.

The *Reset ()* method re-initializes the filter memory to 0.

Program exemple:

```
#include "LinearFilter_Goertzel.hxx"
#include "Noise_Gaussian_MT.hxx"
#include "Noise_GaussianAR.hxx"
#include "Spectral_Spectral.hxx"
#include "Spectral_Taper.hxx"
#include "DCPP_Matrix.hxx"

using namespace _BuL;

////////////////////////////////////////////////////////////////////////////////
int main (int argc, char* argv[])
{
  // variables
  Goertzel filtre (false);

  double samplingFrequency = 20000;
  int dataSize = (int) pow(2.0, 20);
  Vector<double> data (dataSize);
  Vector<double> line (dataSize);
  int k;
  double lineFrequency1 = 100.0;
  double lineFrequency2 = 101.0;
  double filterStartFrequency = 90.0;
  double filterEndFrequency = 110.0;
  double filterStep = 0.01;
  int loopSize = 20000;
  int samplesize = 20000;
  double amplitude = 10;
  double step;
  int index;
  double temp;

  int loop_nb = (filterEndFrequency - filterStartFrequency) / filterStep;

  cout << "data size " << dataSize << endl;
  cout << "loop_nb " << loop_nb << endl;

  Vector<double> window (samplesize);

  Vector<double> resu_1 (loop_nb);
  Vector<double> resu_2 (loop_nb);
  Vector<double> resu_3 (loop_nb);

  Matrix<double> result ((int) (dataSize / loopSize + 1.0),
 (int) ((filterEndFrequency - filterStartFrequency)
/ filterStep + 1.0), 0.0);

  // noise generators
  Gaussian_MT NoiseGen (1.0);

  NoiseGen.GetData (data);

  //add lines
  for (k=0; k < dataSize; k++)
    {
      temp = amplitude * sin (2.0 * M_PI * k * lineFrequency1 / samplingFrequency);
      temp += amplitude * sin (2.0 * M_PI * k * lineFrequency2 / samplingFrequency
       + M_PI/3.0);
      data[k] += temp;
    }

  filtre.Setup (samplingFrequency, lineFrequency1, loopSize);
  cout << "one value " << filtre.Filter (data) << endl;

  Taper hann (samplesize);
  hann.GetDataHann (window);

  // test windowing effect
  //data = 1.0;

  // sweep filter frequency
  index = 0;
  for (step = filterStartFrequency; step <= filterEndFrequency; step += filterStep)
    {
      //cout << "tested frequency " << step << endl;
      filtre.Setup (samplingFrequency, step, loopSize);

      //      filtre.Filter (data, line, window);
      filtre.Filter (data, line);

      for (k=0; k < dataSize / loopSize + 1; k++)
result[k][index] = line[k];

      resu_1[index] = line[0];
      resu_2[index] = line[10];
      resu_3[index] = line[20];

      cout << index << endl;
      index++;
    }

  data.WriteFile ("testdataGoert.dat");
  line.WriteFile ("testlineGoert.dat");
  result.WriteFile ("testsweepGoertzel.dat");
  resu_1.WriteFile ("toto.dat");
  resu_2.WriteFile ("titi.dat");
  resu_3.WriteFile ("tutu.dat");
  return 0;
}
```

2.6 ITF: detector characteristics

This package is intended to contain all the header files defining parameters which are specific to the detectors.
For the moment, it includes all the Earth interferometers header files:

- GEO: "ITF_GEO_Constant.hxx"
- Hanford: "ITF_Hanford_Constant.hxx"
- Livingston: "ITF_Livingston_Constant.hxx"
- TAMA: "ITF_TAMA_Constant.hxx"
- Virgo: "ITF_Virgo_Constant.hxx"
- AIGO: "ITF_AIGO_Constant.hxx"

Their content is given in 2.16.

A generic class called *ITF* is inheritated by several classes which correspond to each Earth based interferometers in activity. The class ITF provides methods to get mainly geometrical information about an interferometer while the inhnerated classes provide a mean to get the power spectral density which is foreseen or measured for each interferometers. The name of the detector is provided by an ITF_Detector enumeration type variable declared in the ITF class and whose definition is given in 2.15 and below:

Detector	Virgo
	Virgo_Quartz
	Virgo_Quartz_Yag
	Hanford_2k
	Hanford_4k
	Livingston
	GEO
	TAMA
	AIGO

The power spectrum density function (one-sided PSD) of Virgo and TAMA interferometer is parametrized as follow:

$$S(f) = \frac{S_{pendulum}}{f^5} + \frac{S_{mirror}}{f} + S_{shotnoise}(1 + \frac{f^2}{f_{cut}})$$

The power spectrum density function of LIGO (same for the three interferometres) and GEO is parametrized as in [16]. Note that the AIGO class does not provide any PSD.

2.6.1 Class ITF

Include file	"ITF_ITF.hxx"
Constructors	ITF()
	ITF(ITF::Detector aName)

Public methods

```
void SetSamplingFrequency(int aValue)
int GetSamplingFrequency()
int GetOneSidedASD(Vector<double> aData)
int GetTwoSidedASD(Vector<double> aData)
int GetOneSidedPSD(Vector<double> aData)
int GetTwoSidedPSD(Vector<double> aData)

int GetOneSidedASD(Vector<double> &aData, double aFactor)
int GetTwoSidedASD(Vector<double> &aData, double aFactor)
int GetOneSidedPSD(Vector<double> &aData, double aFactor)
int GetTwoSidedPSD(Vector<double> &aData, double aFactor)

void PrintOutGeometry()

double SigmaOfOneSidedPSD(Vector<double> aPSD)
double SigmaOfTwoSidedPSD(Vector<double> aPSD)

int WriteOneSidedSpectrumFile(Vector<double> aPSD, string aFileName)
int WriteTwoSidedSpectrumFile(Vector<double> aPSD, string aFileName)

double GetLatitude()
double GetLongitude()
double GetAzimuthal()
double GetArmAngle()
void SetLatitude(double aValue)
void SetLongitude(double aValue)
void SetAzimuthal(double aValue)
void SetArmAngle(double aValue)

void SetDebug(bool aValue)
bool GetDebug()
```

The *ITF()* constructor defines an "empty object" with null geometrical characteristics while the *ITF(ITF::Detector aName)* creates an object filling the sampling information and geometrical characteristics corresponding to interferometer name.

The *SetSamplingFrequency(...)* method allows to (re)define the sampling frequency.
The *GetSamplingFrequency()* method returns the current sampling frequency.

The *GetOneSidedPSD(...)* and *GetOneSidedASD(...)* methods fill the input data vector of size N with the One-sided PSD or ASD. The argument factor allows to get a PSD or ASD which is scaled by this factor. The frequency resolution is given by

$$\delta f = \frac{f_{sampling}}{2N}$$

The *GetTwoSidedPSD(...)* and *GetTwoSidedASD(...)* methods fill the input data vector of size N with the Two Sided PSD or ASD. The argument factor allows to get a PSD or ASD which is scaled by this factor. The frequency resolution is given by

$$\delta f = \frac{f_{sampling}}{N}$$

The *SigmaOfOneSidedPSD(...)* and *SigmaOfTwoSidedPSD(...)* methods compute the sigma of the PSD according to 2.2.

The *WriteOneSidedSpectrumFile(...)* and *WriteTwoSidedSpectrumFile(...)* methods write out in a file the frequency value and the value of a PSD or ASD stored in a vector it corresponds to.

The *GetLatitude()* method returns the latitude of the Virgo detector.
The *GetLongitude()* method returns the longitude of the Virgo detector.
The *GetAzimuthal()* method returns the azimuthal angle of the Virgo detector.
The *GetArmAngle()* method returns the angle between the two arms of the Virgo detector.

The *SetLatitude()* method allows to set the latitude of the Virgo detector.
The *SetLongitude()* method allows to set the longitude of the Virgo detector.
The *SetAzimuthal()* method allows to set the azimuthal angle of the Virgo detector.
The *SetArmAngle()* method allows to set the angle between the two arms of the Virgo detector.

Program example:

```
#include "ITF_ITF.hxx"
using namespace \_BuL;
int main(int arg, char* argc[])
{
  ITF std(ITF::Virgo);
  double longitude = std.GetLongitude();
  double latitude = std.GetLatitude();
  double azimuthal = std.GetAzimuthal();
  double arm_angle = std.GetArmAngle();
  cout << "Virgo Longitude: " << longitude << endl;
  cout << "Virgo Latitude: " << latitude << endl;
  cout << "Virgo azimuthal angle: " << azimuthal << endl;
  return(0) ;
}
```

2.6.2 Class Virgo

This class is inherated from the class ITF, hence all the public methods of ITF are available in the Virgo class.

Include file	"ITF_Virgo.hxx"
Constructors	Virgo()
	Virgo(int aSamplingFrequency)
	Virgo(ITF::Detector aName)
	Virgo(ITF::Detector aName, int aSamplingFrequency)

Public methods

int GetTwoSidedPSD(Vector<double>& aData)
int GetOneSidedPSD(Vector<double>& aData)
int GetTwoSidedASD(Vector<double>& aData)
int GetOneSidedASD(Vector<double>& aData)

int GetOneSidedASD(Vector<double> &aData, double aFactor)
int GetTwoSidedASD(Vector<double> &aData, double aFactor)
int GetOneSidedPSD(Vector<double> &aData, double aFactor)
int GetTwoSidedPSD(Vector<double> &aData, double aFactor)

void SetSamplingFrequency(int aValue)
int GetSamplingFrequency()
double GetLatitude()
double GetLongitude()
double GetAzimuthal()
double GetArmAngle()
void SetLatitude(double aValue)
void SetLongitude(double aValue)
void SetAzimuthal(double aValue)
void SetArmAngle(double aValue)
void SetDebug(bool aValue)
bool GetDebug()

The *Virgo()* constructor consider the ITF::Virgo detector with its nominal f_0=20 kHz sampling frequency. The three other constructors allow to choose another Virgo detector and/or set up a sampling frequency if needed (otherwise the default value is 20 kHz).

The methods are described in the section of the ITF class.

Program exemple:

```
#include "ITF_Virgo.hxx"

using namespace \_BuL;

int main(int arg, char* argc[])
{
  int i, j;
  int size = 10000;
  Vector<double> psd_sdt(size);
  Vector<double> psd_quartz(size);
  Vector<double> psd_quartz2(2*size);
  Virgo std(10000);

  std.GetOneSidedASD(size, psd_sdt);

  for (i=0 ; i<size/1000 ; i++)
    cout << "vector ASD["<< i <<"]= " << psd_sdt[i] << endl;

  Virgo quartz(ITF::Virgo_Quartz, 10000);

  quartz.GetOneSidedASD(size, psd_quartz);

  for (i=0 ; i<size/1000 ; i++)
    cout << "vector ASD["<< i <<"]= " << psd_quartz[i] << endl;

  quartz.GetOneSidedPSD(size, psd_quartz);

  for (i=0 ; i<size/1000 ; i++)
    cout << "vector PSD["<< i <<"]= " << psd_quartz[i] << endl;

  Virgo quartz_bis(ITF::Virgo_Quartz, 20000);

  quartz_bis.GetTwoSidedASD(size, psd_quartz2);

  double longitude = quartz_bis.GetLongitude();
  double latitude = quartz_bis.GetLatitude();
  double azimuthal = quartz_bis.GetAzimuthal();
  double arm_angle = quartz_bis.GetArmAngle();

  cout << "Virgo Longitude: " << longitude << endl;
  cout << "Virgo Latitude: " << latitude << endl;
  cout << "Virgo azimuthal angle: " << azimuthal << endl;
  cout << "Virgo arm angle: " << arm_angle << endl;

  return(0) ;

}
```

2.6.3 Class Hanford

Include file	"ITF_Hanford.hxx"
Constructors	Hanford()
	Hanford(int aSamplingFrequency)
	Hanford(ITF::Detector aName)
	Hanford(ITF::Detector aName, int aSamplingFrequency)

Public methods

int GetTwoSidedPSD(int aSize, Vector<double>& aData)
int GetOneSidedPSD(int aSize, Vector<double>& aData)
int GetTwoSidedASD(int aSize, Vector<double>& aData)
int GetOneSidedASD(int aSize, Vector<double>& aData)

void SetSamplingFrequency(int aValue)
int GetSamplingFrequency()
double GetLatitude()
double GetLongitude()
double GetAzimuthal()
double GetArmAngle()
void SetLatitude(double aValue)
void SetLongitude(double aValue)
void SetAzimuthal(double aValue)
void SetArmAngle(double aValue)
void SetDebug(bool aValue)
bool GetDebug()

The *Hanford()* constructor consider the ITF::Hanford_4k detector with its nominal f_0=16384 Hz sampling frequency.
The three other constructors allow to choose another Hanford detector and/or set up a sampling frequency if needed (otherwise the default value is 16384 Hz).

Note that, for the moment, the spectrum power is identical for the 3 LIGO detectors.

The methods are described in the section of the ITF class.

Program exemple:

```
#include "ITF_Hanford.hxx"
using namespace \_BuL;
int main(int arg, char* argc[])
{
  int i, j;
  int size = 10000;
  Vector<double> psd_2k(size);
  Vector<double> psd_4k(size);

  Hanford H4k(10000);

  H4k.GetOneSidedASD(size, psd_4k);

  for (i=0 ; i<size/1000 ; i++)
```

```
    cout << "vector ASD["<< i <<"]= " << psd_4k[i] << endl;

  cout << " " << endl;

  Hanford H2k(ITF::Hanford_2k, 10000);

  H2k.GetOneSidedASD(size, psd_2k);

  for (i=0 ; i<size/1000 ; i++)
    cout << "vector ASD["<< i <<"]= " << psd_2k[i] << endl;

  double longitude = H2k.GetLongitude();
  double latitude = H2k.GetLatitude();
  double azimuthal = H2k.GetAzimuthal();
  double arm_angle = H2k.GetArmAngle();

  cout << "Hanford Longitude: " << longitude << endl;
  cout << "Hanford Latitude: " << latitude << endl;
  cout << "Hanford azimuthal angle: " << azimuthal << endl;
  cout << "Hanford arm angle: " << arm_angle << endl;

  return(0) ;

}
```

2.6.4 Class Livingston

Include file	"ITF_Livingston.hxx"

Constructors	Livingston()
	Livingston(int aSamplingFrequency)

Public methods

int GetTwoSidedPSD(int aSize, Vector<double>& aData)
int GetOneSidedPSD(int aSize, Vector<double>& aData)
int GetTwoSidedASD(int aSize, Vector<double>& aData)
int GetOneSidedASD(int aSize, Vector<double>& aData)

void SetSamplingFrequency(int aValue)
int GetSamplingFrequency()
double GetLatitude()
double GetLongitude()
double GetAzimuthal()
double GetArmAngle()
void SetLatitude(double aValue)
void SetLongitude(double aValue)
void SetAzimuthal(double aValue)
void SetArmAngle(double aValue)
void SetDebug(bool aValue)
bool GetDebug()

The *Livingston()* constructor consider the ITF::Livingston detector with its nominal f_0=16384 Hz sampling frequency.
The other constructor allow to choose a different sampling frequency if needed.

Note that, for the moment, the spectrum power is identical for the 3 LIGO detectors.

The methods are described in the section of the ITF class.

Program exemple:

```
#include "ITF_Livingston.hxx"
using namespace \_BuL;
int main(int arg, char* argc[])
{
  int i, j;
  int size = 10000;
  Vector<double> psd_sdt(size);

  Livingston std(10000);
  std.GetOneSidedASD(size, psd_sdt);
  for (i=0 ; i<size/1000 ; i++)
    cout << "vector ASD["<< i <<"]= " << psd_sdt[i] << endl;
  return(0) ;
}
```

2.6.5 Class GEO

Include file "ITF_GEO.hxx"

Constructors GEO()
GEO(int aSamplingFrequency)

Public methods

int GetTwoSidedPSD(int aSize, Vector<double>& aData)
int GetOneSidedPSD(int aSize, Vector<double>& aData)
int GetTwoSidedASD(int aSize, Vector<double>& aData)
int GetOneSidedASD(int aSize, Vector<double>& aData)

void SetSamplingFrequency(int aValue)
int GetSamplingFrequency()
double GetLatitude()
double GetLongitude()
double GetAzimuthal()
double GetArmAngle()
void SetLatitude(double aValue)
void SetLongitude(double aValue)
void SetAzimuthal(double aValue)
void SetArmAngle(double aValue)
void SetDebug(bool aValue)
bool GetDebug()

The *GEO()* constructor consider the ITF::GEO detector with its nominal f_0=16384 Hz sampling frequency.
The other constructor allow to choose a different sampling frequency if needed.

The methods are described in the section of the ITF class.

Program exemple:

```
#include "ITF_GEO.hxx"
using namespace \_BuL;

int main(int arg, char* argc[])
{
  int i, j;
  int size = 10000;
  Vector<double> psd_sdt(size);

  GEO std(10000);
  std.GetOneSidedASD(size, psd_sdt);
  for (i=0 ; i<size/1000 ; i++)
    cout << "vector ASD["<< i <<"]= " << psd_sdt[i] << endl;
  return(0) ;
}
```

2.6.6 Class TAMA

Include file	"ITF_TAMA.hxx"
Constructors	TAMA()
	TAMA(int aSamplingFrequency)

Public methods

int GetTwoSidedPSD(int aSize, Vector<double>& aData)
int GetOneSidedPSD(int aSize, Vector<double>& aData)
int GetTwoSidedASD(int aSize, Vector<double>& aData)
int GetOneSidedASD(int aSize, Vector<double>& aData)

void SetSamplingFrequency(int aValue)
int GetSamplingFrequency()
double GetLatitude()
double GetLongitude()
double GetAzimuthal()
double GetArmAngle()
void SetLatitude(double aValue)
void SetLongitude(double aValue)
void SetAzimuthal(double aValue)
void SetArmAngle(double aValue)
void SetDebug(bool aValue)
bool GetDebug()

The *TAMA()* constructor consider the ITF::TAMA detector with its nominal f_0=16384 Hz sampling frequency.
The other constructor allow to choose a different sampling frequency if needed.

The methods are described in the section of the ITF class.

Program exemple:

```
#include "ITF_TAMA.hxx"
using namespace \_BuL;

int main(int arg, char* argc[])
{
  int i, j;
  int size = 10000;
  Vector<double> psd_sdt(size);

  TAMA std(10000);
  std.GetOneSidedASD(size, psd_sdt);
  for (i=0 ; i<size/1000 ; i++)
    cout << "vector ASD["<< i <<"]= " << psd_sdt[i] << endl;
  return(0) ;
}
```

2.6.7 Class AIGO

Include file	"ITF_AIGO.hxx"
Constructors	AIGO()
	AIGO(int aSamplingFrequency)

Public methods

void SetSamplingFrequency(int aValue)
int GetSamplingFrequency()
double GetLatitude()
double GetLongitude()
double GetAzimuthal()
double GetArmAngle()
void SetLatitude(double aValue)
void SetLongitude(double aValue)
void SetAzimuthal(double aValue)
void SetArmAngle(double aValue)
void SetDebug(bool aValue)
bool GetDebug()

The *AIGO()* constructor consider the ITF::AIGO detector with its nominal f_0=xxx Hz sampling frequency.
The other constructor allow to choose a different sampling frequency if needed.

The methods are described in the section of the ITF class.

Program exemple:

```
#include <string>

#include "ITF_AIGO.hxx"

using namespace \_BuL;

int main(int arg, char* argc[])
{
  int i, j;

  AIGO std(10000);

  double longitude = std.GetLongitude();
  double latitude = std.GetLatitude();
  double azimuthal = std.GetAzimuthal();
  double arm_angle = std.GetArmAngle();

  cout << "AIGO Longitude: " << longitude << endl;
  cout << "AIGO Latitude: " << latitude << endl;
  cout << "AIGO azimuthal angle: " << azimuthal << endl;
  cout << "AIGO arm angle: " << arm_angle << endl;

  return(0) ;

}
```

2.7 Noise: generators of white and colored noise

This package includes all classes dedicated for the production of simulated data. The noise models which are considered so far are (this list will be upgraded):

- Gaussian and white
- Gaussian and colored

Two random generators are provided from which are inherated two Gaussian generators. Two models of colored noise are provided: an AR and ARMA model classes. Finally the facilities to produce the AR and ARMA coefficients from a given PSD are provided in this package.

2.7.1 Random generators

2.7.1.1 Class Random_NR

Include file	"Noise_Random_NR.hxx"
Constructors	Random_NR()
Public methods	

```
void GetRandomData (Vector<double>& aData)
void GetRandomData (Vector<float>& aData)
double GetRandomData ()
double GetRandomData (double aXmin, double aXmax)
double GetGaussianData ()

void SetSeed ()
void SetSeedLocalTime ()

void SetDebug (const bool aValue)
bool GetDebug ()
```

This Random_NR class provides facilities to generate a random variable whose values are in between a given segment $[x_{min}, x_{max}]$. The random variable algorithm used in this class is based on the drand48() function [11]. The internal seed number is initialized to 0.

The *SetSeed()* allows to increment the internal seed number of 1 unit. This method can be called at any instant.

The *SetSeedLocalTime()* allows to increment the seed number used by drand48() of a given number which depends on the local time (with a precision of 1 s). The internal seed number is also increment of one unit. This method can be called at any instant.

The *GetRandomData()* returns a random variable whose value lays in the segment $[0, 1]$. Note that there is no change of a new sequence seed at each call of the function.

The *GetRandomData(aXmin, aXmax)* returns a random variable whose value lays in $[aXmin, aXmax]$. Note that there is no change of a new sequence seed at each call of the function.

The *GetRandomData(Vector<T> aData)* fills a vector with random variable whose value lays in the segment $[0, 1]$. **Note that, contrary to the previous functions, there is a change of a new sequence seed at each call of the function. For that purpose, the method SeedLocalTime() is used**

The *GetGaussianData()* methods returns a Gaussian random variable whose value lays in the segment $[0, 1]$. The Gaussian variable is built from the random variable using the Box Muller algorithm described in [11]. Note that there is no change of a new sequence seed at each call of the function.

2.7.1.2 Class Random_MT

Include file	"Noise_Random_MT.hxx"
Constructors	Random_MT()
Public methods	
	void GetRandomData (Vector<double>& aData)
	void GetRandomData (Vector<float>& aData)
	double GetRandomData ()
	double GetRandomData (double aXmin, double aXmax)
	double GetGaussianData ()
	void SetSeed (unsigned long aSeed)
	void SetSeedLocalTime (unsigned long aSeed)
	void SetDebug (bool aValue)
	bool GetDebug ()

This Random_MT class provides facilities to generate a random variable whose values are in between a given segment $[x_{min}, x_{max}]$. The random variable algorithm used in this class is the Mersenne Twister algorithm [12]. Its period is $2 * *19937 - 1$ and it is 5 times faster than the previous Random_NR generator.
The internal seed number is initialized to 0.

The *SetSeed(...)* allows to set the internal seed number to the given value. This method can be called at any instant.

The *SetSeedLocalTime(...)* allows to set the seed number used by MT algorithm of a given number which depends on the local time (with a precision of 1 s). The internal seed number is set to the value given as argument. This method can be called at any instant.

The *GetRandomData()* returns a random variable whose value lays in the segment $[0, 1]$. Note that there is no change of a new sequence seed at each call of the function.

The *GetRandomData(aXmin, aXmax)* returns a random variable whose value lays in $[aXmin, aXmax]$. Note that there is no change of a new sequence seed at each call of the function.

The *GetRandomData(Vector<double> aData)* fills a vector with random variable whose value lays in the segment $[0, 1]$. **Note that, contrary to the previous functions, there is a change of a new sequence seed at each call of the function. For that purpose, it uses the SetSeedLocalTime() method.**

The *GetGaussianData()* method returns a Gaussian random variable whose value lays in the segment $[0, 1]$. The Gaussian variable is built from the random variable using the Box Muller algorithm described in [11]. Note that there is no change of a new sequence seed at each call of the function.

2.7.2 Gaussian and white generators

The two following classes allow to generate a white and gaussian distribution noise. By default, the mean of the distribution is zero and the sigma σ_n is derived from the Virgo performance values stored in the header file "ITF_Constant.hxx" located in the BuL/ITF package (one assumes that the spectral power density of Virgo is equal to its minimal value h_n^2):

f_s = 20 kHz : sampling frequency
$S_h(f)_{min} = h_n = \frac{2\sigma_n^2}{f_s}$: minimal value of the one-sided power spectral density
$h_n = 4.\ 10^{-23}/\sqrt{Hz}$: VIRGO minimal sensitivity noise
$\sigma_n = \frac{h_n \sqrt{f_s}}{\sqrt{2}} = 4.\ e^{-21}$: VIRGO noise sigma for bursts

2.7.2.1 Class Gaussian_NR

Include file	"Noise_Gaussian_NR"	
Constructors	Gaussian_NR()	I
	Gaussian_NR(double aSigma)	II
Public methods	(in addition to the one of the class Random_NR)	
	int GetData(Vector<double>& aData)	
	int GetData(Vector<float>& aData)	
	double GetValue()	
	int SetSamplingFrequency(int aValue)	
	int GetSamplingFrequency()	
	int SetGaussianSigma(double aSigma)	

The class Gaussian_NR is making use of the Random_NR class to generate random numbers.

The first constructor initializes the sigma of the white and Gaussian noise to the "official" Virgo value. The second constructor allows to specify the sigma value.

The *GetData(...)* methods fill up a vector of the given size of values generated according to a white noise random distribution. Note that the memory allocation of the vector has to be done by the user. This method makes use of the Random_NR::GetGaussianData. Before calling this method GetData changes the seed using SetSeedLocalTime() in order to avoid problem.

The *GetValue()* method returns a scalar value generated according to a white noise random distribution. The condition are identical to *GetData()* method.

The *SetSamplingFrequency(...)* method changes the sampling frequency private variable and also the sigma value of the noise according to the formula given above. The sampling frequency is used only in the computation of the sigma when one uses the constructor I.

The *GetSamplingFrequency(...)* method gets the value of the sampling frequency used.
On the contrary, the *SetGaussianSigma(...)* method changes only the value of the sigma of the noise.

2.7.2.2 Class Gaussian_MT

Include file	"Noise_Gaussian_MT"	
Constructors	Gaussian_MT()	I
	Gaussian_MT(double aSigma)	II
Public methods	(in addition to the one of the class Random_MT)	
	int GetData(Vector<double>& aData)	
	int GetData(Vector<float>& aData)	
	double GetValue()	
	int SetSamplingFrequency(int aValue)	
	int GetSamplingFrequency()	
	int SetGaussianSigma(double aSigma)	

The class Gaussian_MT is making use of the Random_MT class to generate random numbers.

The first constructor initializes the sigma of the white and Gaussian noise to the "official" Virgo value. The second constructor allows to specify the sigma value.

The *GetData(...)* methods return a vector of the given size of values generated according to a white noise random distribution. Note that the memory allocation of the vector has to be done by the user.
This method makes use of the Random_MT::GetGaussianData() and hence there is no change of Seed number before generating data. It is up to the user to do it by calling *SetSeed()* or *SetSeedLocalTime()* methods.

The *GetValue()* method returns a scalar value generated according to a white noise random distribution. The conditions are identical to the *GetData()* method.

The *SetSamplingFrequency(...)* method changes the sampling frequency private variable and also the sigma value of the noise according to the formula given above. The sampling frequency is used only in the computation of the sigma when one uses the constructor I.

The *GetSamplingFrequency(...)* method gets the value of the sampling frequency used.
On the contrary, the *SetGaussianSigma(...)* method changes only the value of the sigma of the noise.

Program example:

```
#include "Noise_Gaussian_MT.hxx"
#include "DataContainer_Vector.hxx"
#include "Spectral_Spectral.hxx"
using namespace \_BuL;
int main(int arg, char* argc[])
{
  int size = 2000000;
  int mean = 10;
  Vector<double> data(size);
  Vector<double> out(size/2./mean);

  Gaussian_MT *gene = new Gaussian_MT();
  gene->SetGaussianSigma(1.);
  gene->GetData(data);
  data.WriteFile("white_gaussian.dat");
  Spectral spec(size/mean, Taper::Haming, mean, Spectral::OneSided);
  spec.ASD(data, out, 20000);
  spec.WriteSpectrumFile(out, "psd.dat", 20000);
  return(0) ;
}
```

2.7.3 Gaussian and colored generators

The 3 following classes provide different facilities to generate a Gaussian and Colored noise, according to different model. The Two first classes supposed that the noise can be modelized by a Auto-Regressive (AR) or Auto-Regressive Moving-Average (ARMA) models[14, 15, 13]. The last generator performs the convolution of a white and Gaussian noise with a "theoritical" processus whose PSD is known. This method, while efficient, requests to perform 2 Fourier Transforms (time consuming) and is not able to produce a continuous chunk of colored and Gaussian data since the phase is not set properly.

2.7.3.1 Class GaussianAR

Include file	"Noise_GaussianAR_hxx"	
Constructors	GaussianAR (string aFileName)	I
	GaussianAR (ITF::Detector aITF)	II

Public methods

```
int GetData (Vector<double>& aData)

int GetSamplingFrequency ()
int GetPolesNumber ()
double GetSigma ()

int GetPSDFromData (double aDeltaFrequency, int aAverageNumber,
                    Vector<double> &aPSD)
int GetPSDFromCoefficients ( double aDeltaFrequency, Vector<double> &aPSD)

void SetSeed (unsigned long aSeed)
void SetSeedLocalTime (unsigned long aSeed)

void SetDebug (bool aValue)
bool GetDebug ()
```

This class provides a facility to generate a Gaussian Noise whose PSD is parametrized by a AR filter model[13].

Two constructors are provided: for the first one (I) the user must give the file of the AR coefficients which correspond to the PSD of the data he wants to simulate (to know how to determine such coefficients see section 2.7.4). In the second constructor (II) the user gives the name of the interferometer wich corresponds to the data he wants to generate (All the AR coefficients files corresponding to each "official" interferometer PSD curves have been produced).
The format of the AR coefficients file is the following: A comment line is indicated with

```
# Info
```

The Sigma value is the discrete sigma value.
The Sampling value gives the sampling frequency of the process.
The Frequency info line gives the minimal frequency which is simulated. Below this frequency the spectrum is flat. The AR value gives the number P of poles of the AR filter. The coefficients values are then given at the end of the file.

```
# Sigma 1.5900433817564251e-20
# Sampling 20000
# Info Frequency cut 5.00
# AR 512
1    9.0243985366985069e-01
2    5.9914915520694234e-01

...

511    -9.9890829891022292e-04
512    -5.2225395422075837e-04
```

The *GetData(...)* method fills a vector with Gaussian and colored data followed:

$$y[i] = -\sum_{k=1}^{P} a_k y[i-k] + Sigma \times w[i]$$

where $w[i]$ is a white and Gaussian random variable of variance equal to 1 and a_k are the AR coefficients.

The *GetPSDFromData (...)* method estimates the one-sided PSD of a vector of generated data using the AR parameters of the class.

The *GetPSDFromCoefficients (...)* method computes the one-sided PSD numerically using the AR parameters following:

$$PSD(f) = 2 \times Sigma^2 \times \frac{1}{|1 + \sum_{k=1}^{P} a_k e^{-2\pi i f k}|^2}$$

The *GetSamplingFrequency()* returns the Sampling frequency used in the AR process.

The *GetPolesNumber()* returns the number of poles (number of AR coefficients) of the AR process.

The *GetSigma()* returns the value of the discrete sigma which is used in the AR process.

The *SetSeed(...)* allows to set the internal seed number to the given value. This method can be called at any instant.

The *SetSeedLocalTime(...)* allows to set the seed number used by MT algorithm of a given number which depends on the local time (with a precision of 1 s). The internal seed number is set to the value given as argument. This method can be called at any instant.

Program exemple:

```
#include "Noise_GaussianAR.hxx"
#include "DataContainer_Vector.hxx"
#include "Spectral_Spectral.hxx"

using namespace \_BuL;

int main(int arg, char* argc[])
{
  int i, j;
  int sampling = 20000;
  int size = sampling/2*10;
  int mean = 10;
  Vector<double> data(mean*size);
  Vector<double> out(size/2.);
  string ar_coeff;

  ar_coeff = "../data/AR_Virgo_1024_4Hz.dat";

  GaussianAR gene(ar_coeff);
  gene.SetDebug(false);

  gene.GetData(data);
  gene.GetData(data);

  for (i=0; i< 10 ; i++)
    cout << data[i] << endl;

  data.WriteFile("data.dat");

  Spectral spec(size, Taper::Haming, mean, Spectral::OneSided);

  spec.ASDWOSA(data, out, sampling);

  spec.WriteSpectrumFile(out, "psd.dat", sampling);

  return(0) ;

}
```

2.7.3.2 Class GaussianARMA

Include file	"Noise_GaussianARMA_hxx"	
Constructors	GaussianARMA(string aFileName)	I
	GaussianARMA(ITF::Detector aITF)	II

Public methods

```
int GetData(Vector<double>& aData)

int GetPSDFromData (double aDeltaFrequency, int aAverageNumber,
                    Vector<double> &aPSD)
int GetPSDFromCoefficients ( double aDeltaFrequency, Vector<double> &aPSD)

int GetSamplingFrequency()
int GetPolesNumber()
double GetSigma()

void SetSeed (unsigned long aSeed)
void SetSeedLocalTime (unsigned long aSeed)

void SetDebug(bool aValue)
bool GetDebug()
```

This class provides a facility to generate a Gaussian Noise whose PSD is parametrized by a ARMA filter model[13].

As for the GaussianAR class, two constructors are provided: for the first one (I) the user must give the file of the ARMA coefficients which correspond to the PSD of the data he wants to simulate (to know how to determine such coefficients see section 2.7.5). In the second constructor (II) the user gives the name of the interferometer wich corresponds to the data he wants to generate (All the ARMA coefficients files corresponding to each "official" interferometer PSD curves have been produced).
The format of the ARMA coefficients file is the following: A comment line is indicated with

```
# Info
```

The Sigma value is the discrete sigma value.
The Sampling value gives the sampling frequency of the process.
The Frequency info line gives the minimal frequency which is simulated. Below this frequency the spectrum is flat. The AR value gives the number P of poles of the ARMA filter. The AR coefficient values are then given.
The MA value gives the number Q of zeros of the ARMA filter. The MA coefficient values are then given.

```
# Sigma 1.8522732942547901e-20
# Sampling 20000
# Info Frequency cut 10.00
# AR 38
1    -3.3181070132757395e-01
2    -3.7689826182215138e+00
...
38    -2.6785776758171774e-11
# MA 16
0    8.5722580329088460e-01
1    -1.0604343586542779e+00
```

```
2    -2.7890994776648230e+00
...
16    2.3504054468144381e-02
```

The *GetData(...)* method fills a vector with Gaussian and colored data followed:

$$y[i] = -\sum_{k=1}^{P} a_k y[i-k] + Sigma \times \sum_{k=0}^{Q} b_k w[i]$$

where $w[i]$ is a white and Gaussian random variable of variance equal to 1, a_k are the AR coefficients and b_k are the MA coeffients.

The *GetPSDFromData (...)* method estimates the onde-sided PSD of a vector of generated data using the ARMA parameters of the class.

The *GetPSDFromCoefficients (...)* method computes the one-sided PSD numerically using the ARMA parameters following:

$$PSD(f) = 2 \times Sigma^2 \times \frac{|\sum_{k=0}^{Q} b_k e^{-2\Pi ifk}|^2}{|1 + \sum_{k=1}^{P} a_k e^{-2\Pi ifk}|^2}$$

The *GetSamplingFrequency()* returns the Sampling frequency used in the AR process.

The *GetPolesNumber()* returns the number of poles (number of AR coefficients) of the AR process.

The *GetSigma()* returns the value of the discrete sigma which is used in the AR process.

The *SetSeed(...)* allows to set the internal seed number to the given value. This method can be called at any instant.

The *SetSeedLocalTime(...)* allows to set the seed number used by MT algorithm of a given number which depends on the local time (with a precision of 1 s). The internal seed number is set to the value given as argument. This method can be called at any instant.

2.7.3.3 Class GaussianColored

Include file "Noise_GaussianColored.hxx"

Constructors GaussianColored(int aSize, ITF::Detector aITF)

Public methods

int GetData(Vector<double> & aData)
int GetData(Vector<float> & aData)
int SetSamplingFrequency(int aValue)
int GetSamplingFrequency()
void WriteFileGeneratedASD(string aName)
void WriteFileGeneratedASD(int aSize, string aName)
double GetValue()

2.7.4 Class AR

Include file	"Noise_AR.hxx"
Constructors	AR (int aSampling, int aPolesNumber)
Public methods	

```
int ComputeAR (Vector<double>& aPSD, double aPSDScaleFactor,
               double aMinimalFrequency)
int ComputeAR (ITF::Detector aITF, double aMinimalFrequency)
int SaveAR (string aFileName)
double GetSigma (int aIndex)
double GetAR (int aIndex)
void GetAR (Vector<double>& aAR)

int SetDeltaFrequency (double aValue)
double GetDeltaFrequency ()
int SetPolesNumber (int aValue)
int GetPolesNumber ()
int GetSamplingFrequency ()
int SetSamplingFrequency (int aSampling)

int FindOptimalOrderMDL (ITF::Detector aITF, double aMinimalFrequency,
                         int aDataSize)

void SetDebug (bool aValue)
bool GetDebug ()
```

The *ComputeAR(...)* method compute the AR coefficients from the PSD given as argument using the Durbin-Levinson regression to solve the Yule-Walker equations [13]. The *aPSDScaleFactor* argument is the scale factor which has been applied to the PSD in order to avoid to manipulate very small number. The *aMinimalFrequency* argument is the frequency below which the PSD is taken as constant and equal to the value at *aMinimalFrequency* .

The *ComputeAR(...)* method compute the AR coefficients taking into account the "official" PSD which corresponds to the given interferometer. The *aMinimalFrequency* argument is the frequency below which the PSD is taken as constant and equal to the value at *aMinimalFrequency* .

The *SaveAR(...)* method saves in the file whose name is given as argument, the AR filter coefficients according to the format which is described in section 2.7.3.1.

The *GetSigma(aIndex)* method returns the value of the P estimations of the sigma value. The Sigma which is used in the AR model is the one corresponding to the index=P. Note that the aIndex value varies from 0 to P.

The *GetAR(...)* methods return the values of the coefficients of the AR filter.

The *SetDeltaFrequency(...)* and *GetDeltaFrequency(...)* methods allow to change and to get the value of the frequency step which is used when computing the autocorrelation function.

The *SetPolesNumber(...)* and *GetPolesNumber(...)* methods allow to change and get the number of poles of the AR filter.

The *SetSamplingFrequency(...)* and *GetSamplingFrequency(...)* methods allow to change and get the number of poles of the AR filter.

The *FindOptimalOrder(...)* method determine for a give interferomter the optimal number of poles according to the ML criteria (to be explained).

Program example:

```
#include "Noise_AR.hxx"
#include "DataContainer_Vector.hxx"
#include "Spectral_Spectral.hxx"
#include "ITF_Virgo.hxx"
using namespace \_BuL;
int main(int arg, char* argc[])
{
  int poles_number, sampling, size;
  double f_cut = 4.;
  double PSD_scale_factor = 1.e40;

  poles_number = 1024;
  sampling = 20000;
  size = sampling/2;

  AR algo(sampling, poles_number);
  algo.SetDebug(false);

  // precision de 0.01 Hz
  Vector<double> PSD(100*size);

  // on genere la PSD
  Virgo itf(sampling);
  itf.GetOneSidedPSD(PSD);

  // on calcule les coeff a partir de la PSD
  algo.ComputeAR(PSD, PSD_scale_factor, f_cut);

  cout << "sigma: " << sqrt(algo.GetSigma(poles_number)) << endl;

  algo.SaveAR("AR1.dat");

  // on calcule les coeff directement a parti du nom du detecteur
  algo.SetDeltaFrequency(1.);
  algo.ComputeAR(ITF::Virgo, f_cut);

  cout << "sigma: " << sqrt(algo.GetSigma(poles_number)) << endl;

  algo.SaveAR("AR2.dat");

// To determine the optimla value of the poles_number
  algo.SetDeltaFrequency(.1);
  int opt = algo.FindOptimalOrderMDL(ITF::Virgo, f_cut, 20000);

  cout << " Valeur optimale de l ordre selon le MDL critere: " << opt << endl;
  return(0) ;
}
```

2.7.5 Class ARMA

Exliquer ici la procedure de resolution des equations Normales.

Include file	"Noise_ARMA.hxx"
Constructors	ARMA (int aSampling, int aPolesNumber, int aZerosNumber)
Public methods	

```
int ComputeARMA (Vector<double>& aPSD, double aPSDScaleFactor,
                              double aMinimalFrequency)
int ComputeARMA (ITF::Detector aITF, double aMinimalFrequency)

int SaveARMA (string aFileName)
void GetAR (Vector<double> & aAR)
void GetMA (Vector<double> & aMA)

int GetSamplingFrequency ()
int SetSamplingFrequency (int aSampling)
int SetDeltaFrequency (double aValue)
double GetDeltaFrequency ()
int SetPolesNumber (int aPolesNumber)
int SetZerosNumber (int aZerosNumber)
int GetPolesNumber ()
int GetZerosNumber ()
```

The *ComputeARMA(...)* method computes the AR and MA coefficients from the PSD given as argument using the procedure described above. The *aPSDScaleFactor* argument is the scale factor which has been applied to the PSD in order to avoid to manipulate very small number. The *aMinimalFrequency* argument is the frequency below which the PSD is taken as constant and equal to the value at *aMinimalFrequency.*

The *ComputeAR(...)* method computes the AR and MA coefficients as the previous method but taking into account the "official" PSD which corresponds to the given interferometer. The *aMinimalFrequency* argument is the frequency below which the PSD is taken as constant and equal to the value at *aMinimalFrequency* .

The *SaveARMA(...)* method saves in the file whose name is given as argument, the ARMA filter coefficients according to the format which is described in section 2.7.3.2.

The *GetAR(...)* and *GetMA(...)* methods return the values of the coefficients of the AR and MA part of the ARMA filter.

The *SetDeltaFrequency(...)* and *GetDeltaFrequency(...)* methods allow to change and to get the value of the frequency step which is used when computing the autocorrelation function.

The *SetPolesNumber(...)* and *GetPolesNumber(...)* methods allow to change and get the number of poles of the AR filter.

The *SetZerosNumber(...)* and *GetZerosNumber(...)* methods allow to change and get the number of poles of the AR filter.

The *SetSamplingFrequency(...)* and *GetSamplingFrequency(...)* methods allow to change and get the number of poles of the AR filter.

Program example:

```
#include "Noise_ARMA.hxx"
#include "DataContainer_Vector.hxx"
#include "Spectral_Spectral.hxx"
#include "ITF_Virgo.hxx"

using namespace \_BuL;
int main(int arg, char* argc[])
{
  int i, j, sampling, size;
  int poles_nb, zeros_nb;
  double f_cut;

  sampling = 20000;
  poles_nb = 38;
  zeros_nb = 16;
  size = sampling/2;
  f_cut = 10.;
  double factor = 1.e40;

  // Virgo PSD
  Vector<double> PSD(100*size);   // precision de 0.01 Hz
  Virgo itf(sampling);

  itf.GetOneSidedPSD(PSD);

  ARMA model(sampling, poles_nb, zeros_nb);

  model.ComputeARMA(PSD, factor, f_cut);

  model.SaveARMA("test.dat");

  model.ComputeARMA(ITF::Virgo, f_cut);
  model.SaveARMA("test2.dat");

  return(0) ;
}
```

2.7.6 Class GaussianLine

2.7.6.1 Class Line

Include file	**Noise_Line.hxx**

Constructors

```
Line ()
Line (bool aDebug)
Line (int aSampling, double aFrequency, double aQ, double aMass,
      double aTemperature, double aCoefficient)
Line (int aSampling, double aFrequency, double aSigmaProcess,
      double aCoefficient)
Line (Line &aLine)
```

Destructors

```
~Line ()
```

Public methods

```
int SetThermal (int aSamplingFrequency,double aFrequency, double aQ,
                double aMass, double aTemperature, double aCoefficient)

int SetThermal (double aFrequency, double aQ,double aMass,
                double aTemperature, double aCoefficient)
int SetElectronic (int aSamplingFrequency, double aFrequency,
                   double aCoefficient)
int SetElectronic (double aFrequency, double aSigmaProcess,
                   double aSigmaProcess, double aCoefficient)
int SetSamplingFrequency (int aSampling)

void SetDebug (bool aValue)
bool GetDebug ()
int GetProcessNoise (_Virgo::Vector<double>& aProcessNoise)

double GetSigmaProcess ()

double GetSigmaProcess ()
double GetCoefficient ()

double GetOmega ()
double GetMass ()
double GetTemperature ()
double GetQuality ()
```

2.7.6.2 Class GaussianLine

Include file **Noise_GaussianLine.hxx**

Constructors

```
GaussianLine ()
GaussianLine (bool aDebug, int aSamplingFreq)
GaussianLine (string aFileName)
```

Destructors

```
~GaussianLine ()
```

Public methods

```
int Reset (double aF, double aQ, double aMass, double aT,
           double aSigmaProcessCoef, double aSigmaMeasure,
           double aMeasureCoef)
int Reset (int aSamplingFreq, double aF, double aQ, double aMass,
           double aT, double aSigmaProcessCoef, double aSigmaMeasure,
           double aMeasureCoef)
int Reset (void)

int GetData (Vector<double>& aData)

void SetDebug (bool aValue)
bool GetDebug ()
```

2.8 BuS: sources of Burst events

This package includes one generic virtual C++ class (BuS_Source) from which is five other classes (BuS_Gaussian, BuS_DampedSine, BuS_GaussianSine, BuS_ZM, and BuS_DFM) inherit.

The five previous classes provide methods to fill Vector of data with Gaussian, damped sine and Gaussian sine waveforms as well as with the result of the core collapse simulation performed by Zwerger and Mueller (ZM catalogue [17]) and Dimmelmeier, Font and Mueller (DFM catalogue [19]). The parameters of each signal type are given when creating the object and can be modified later on, if needed. For what concerns the normalization of the waveform, one has to give a Signal to Noise Ratio (SNR) value or a distance value but only for core collapse waveforms (ZM and DFM). When dealing with SNR normalization, the type of noise hypothesis is important. The choice between "white noise" or "colored noise" hypotheses is done at the level of the constructor and can be modified. The following table summarizes the different use cases:

Signal	Normalization	Noise	Constructor
Gaussian	SNR	White Noise	I
Gaussian	SNR	Colored Noise	II & III
DampedSine	SNR	White Noise	I
DampedSine	SNR	Colored Noise	II & III
GaussianSine	SNR	White Noise	I
GaussianSine	SNR	Colored Noise	II & III
ZM	SNR	White Noise	I
ZM	SNR	Colored Noise	III & V
ZM	distance	White Noise & Colored Noise	II & IV & VI
DFM	SNR	White Noise	I
DFM	SNR	Colored Noise	III & V
DFM	distance	White Noise & Colored Noise	II & IV & VI

The SNR is defined as [21]:

$$SNR^2 = 2\int_{-\infty}^{+\infty} \frac{|\tilde{h}|^2 df}{S_h(f)} = 4\int_{0}^{+\infty} \frac{|\tilde{h}|^2 df}{S_h(f)}$$

where $S_h(f)$ is the one-sided power spectral density function of the interferometer.
In the case of a White Noise hypothesis the SNR is simply:

$$SNR^2 = \frac{f_s}{\sigma_n^2}\int_{-\infty}^{+\infty} |\tilde{h}|^2 df = \frac{f_s}{\sigma_n^2}\int_{-\infty}^{+\infty} |h(t)|^2 dt$$

2.8.1 Class Source

The class Source is a generic class containing virtual methods which are implemented in the derived classes, and hence one cannot instance a Source object. Nevertheless, the Source class contains some public methods which are directly inherated by the derived classes. This is the reason why we give the list of the public methods of the Source class in this user's manual.

When one creates a specific source object, one has to specify the noise model which is supposed to be used using the appropriate constructor.
Default noise caracteristics when a Source object is created:

- White Noise:
  The sampling frequency and the white noise sigma are extracted from the "Virgo_Constant.hxx" header file which values are given in sections 2.7.2 and 2.6.
- Colored Noise:
  One gives the PSD in an input Vector or one gives the name of the interferometer whose PSD and sampling frequency are taken from 2.16.

Here is a list of the methods which can be used in any specific source class.

Public methods

```
int SetDebug (bool aValue)
bool GetDebug ()
int GetSamplingFrequency ()
int SetSamplingFrequency (int aValue)
int SetSigmaNoise (double aSigma)
double GetSigmaNoise ()
int SetSNR (double aSNR)
int SetPSD (int aSampling, Vector<double> aPSD)
int SetITF (ITF::Detector aName)
```

- The following methods have action only in the case of a white noise:

  The method *SetSamplingFrequency(...)* changes the value of the sampling frequency. The value of the sigma of the white noise is also changed according to the formula given in section 2.7.2.
  The normalization factor (in the case of SNR normalization) is recomputed.

  The method *SetSigmaNoise(...)* method changes the value of the white noise sigma in the case of a white noise option choice. The normalization factor (in the case of SNR normalization) is recomputed.

  The method *GetSigmaNoise(...)* method returns the value of the sigma of the white noise.

- The following methods are active only in the case of the "colored noise" option:

  The method *SetPSD(...)* changes the definition of the PSD and the sampling frequency.

  The method *SetITF(...)* changes the definition of the interferometer whose noise characteristics are taken into account for the source normalization.

- The following methods are active in the case of the both noise options:

  The method *GetSamplingFrequency()* returns the sampling frequency.

  The method *SetSNR(...)* changes the value of the SNR. The normalization factor is then recomputed.

2.8.2 Class Gaussian

The class Gaussian provides facilities to generate a waveform:

$$f(t) = \alpha e^{\frac{-t^2}{2\sigma^2}}$$

Include file	"BuS_Gaussian.hxx"
Constructors	Gaussian (double aSigma, double aSNR) Gaussian (double aSigma, double aSNR, int aSampling, Vector<double> &aPSD) Gaussian (double aSigma, double aSNR, ITF::Detector aName)
Public methods	int GetData (double aPosition, Vector<double>& aData) int GetData (double aPosition, Vector<float>& aData) double GetValue (double aX) int SetSigma (double aSigma) int SetDistance (int aDistance)

The constructors define the σ parameter of the Gaussian function, and the SNR value which is used to compute the normalization factor which takes into account the kind of noise which is chosen and the SNR value.

The method *GetData(...)* fills the data vector with a gaussian signal which is located at the relative position given by the argument ***aPosition***. This position is a fraction of the data vector size.

The method *GetValue(...)* returns the discrete value of the Gaussian waveform.

The method *SetSigma(...)* changes the value of the sigma of the Gaussian waveform.

The method *SetDistance(...)* performs no action.

Program example:

```
#include "BuS_Gaussian.hxx"
using namespace \_BuL;
int main(int arg, char* argc[])
{
  Vector<double> dou(size);
  double position = .5;
  double sigma = .1;
  double SNR = 10.;

  Gaussian signal(sigma, SNR);
  signal.SetSigmaNoise(1.);
  signal.GetData(position, dou);
  for (int j=size/2 ; j<size/2+2 ; j++)
    cout << "Vector dou["<< j <<"]= " << dou[j] << endl;
  ITF antenne(ITF::Virgo);
  Vector<double> PSD(1000000);
  antenne.GetOneSidedPSD(PSD);
  Gaussian signal_color(sigma, SNR, sampling, PSD);
  signal_color.GetData(position, dou);
  for (int j=size/2 ; j<size/2+2 ; j++)
    cout << "Vector dou["<< j <<"]= " << dou[j] << endl;
  return(0) ;
}
```

2.8.3 Class DampedSine

The class DampedSine provides facilities to generate a waveform:

$$f(t) = \alpha e^{-\frac{t}{\tau}} sin(2\pi f t)$$

and

$$f(t) = \alpha e^{-\frac{t}{\tau}} cos(2\pi f t)$$

Include file	**BuS_DampedSine.hxx**
Constructors	`DampedSine (double aTau, double aFrequency, int aDampedType, double aSNR)`
	`DampedSine (double aTau, double aFrequency, int aDampedType, double aSNR, int aSampling, Vector<double> &aPSD)`
	`DampedSine (double aTau, double aFrequency, int aDampedType, double aSNR, ITF::Detector aName)`
Destructors	`~DampedSine ()`
Public methods	`int GetData (double aPosition, Vector<double> aData)`
	`int GetData (double aPosition, Vector<float> aData)`
	`double GetValue (double aX)`
	`int SetTau (double aTau)`
	`int SetFrequency (double aFrequency)`
	`int SetDistance (int aDistance)`
	`int SetDamped (int aDampedType)`

The constructors define the τ and f parameters of the DampedSine function, and the SNR value which is used to compute the normalization factor which takes into account the kind of noise which is chosen and the SNR value. The *aDamped* argument defines the chosen quadrature as follow:

- 0: cosine
- 1: sine

Remind that one has to give the one sided PSD in case of a colored noise.

The method *GetData(...)* fills the data vector with a DampedSine signal which is located at the relative position given by the argument *aPosition.* This position is a fraction of the data vector size.

The method *GetValue(...)* returns the discrete value of the DampedSine waveform.

The method *SetTau(...)* changes the value of the τ of the DampedSine waveform.

The method *SetFrequency(...)* changes the value of the f parameter of the DampedSine waveform.

The method *SetDistance(...)* performs no action.

The method *SetDamped (...)* allows to switch from one quadrature to anothre at any time. The change is taken into account at the next call of *GetData ()*.
Program example:

```
#include "BuS_DampedSine.hxx"
using namespace \_BuL;
int main(int arg, char* argc[])
{
  Vector<double> dou(size);
  double position = .25;
  double tau = 0.005;
  double frequency = 300;
  double SNR = 10.;
  int damped = 1;

  DampedSine signal(tau, frequency, damped, SNR);
  signal.GetData(position, dou);
  for (int j=size/4 + 16 ; j<size/4 + 19 ; j++)
    cout << "Vector dou["<< j <<"]= " << dou[j] << endl;
  DampedSine signal_itf(tau, frequency, 10, ITF::Virgo);
  signal_itf.SetSNR(100.);
  signal_itf.GetData(position, dou);
  for (int j=size/2 ; j<size/2+2 ; j++)
    cout << "Vector dou["<< j <<"]= " << dou[j] << endl;
  return(0) ;
}
```

2.8.4 Class GaussianSine

The class GaussianSine provides facilities to generate a waveform:

$$f(t) = \alpha e^{-\frac{t^2}{2\sigma^2}} sin(2\pi f t)$$

Include file "BuS_GaussianSine.hxx"

Constructors GaussianSine (double aSigma, double aFrequency, double aSNR)
GaussianSine (double aSigma, double aFrequency, double aSNR, int aSampling,
Vector<double> &aPSD)
GaussianSine (double aSigma, double aFrequency, double aSNR, ITF::Detector aName)

Public methods

int GetData (double aPosition, Vector<double>& aData)
int GetData (double aPosition, Vector<float>& aData)
double GetValue (double aX)
int SetSigma (double aSigma)
int SetFrequency (double aFrequency)

int SetDistance (int aDistance)

The constructors define the σ and f parameters of the GaussianSine function, and the SNR value which is used to compute the normalization factor which takes into account the kind of noise which is chosen and the SNR value.

The method *GetData(...)* fills the data vector with a GaussianSine signal which is located at the relative position given by the argument *aPosition.* This position is a fraction of the data vector size.

The method *GetValue(...)* returns the discrete value of the GaussianSine waveform.

The method *SetSigma(...)* changes the value of the σ of the GaussianSine waveform.

The method *SetFrequency(...)* changes the value of the f parameter of the GaussianSine waveform.

The method *SetDistance(...)* performs no action.

Program example:

```
#include "BuS_GaussianSine.hxx"
using namespace \_BuL;
int main(int arg, char* argc[])
{
  double position = .5;
  double sigma = 0.1;
  double frequency = 300;
  double SNR = 10.;

  GaussianSine signal(sigma, frequency, SNR);
  signal.SetSigmaNoise(1.);
  signal.SetSNR(100.);
  signal.SetSamplingFrequency(10000);

  double x = 1./4000.;
  cout << signal.GetValue(-x) << endl ;
}
```

2.8.5 Class ZM

The Zwerger & Mueller catalogue gathers a collection of waveforms produced by the Newtonian simulation of the collapse of a rotating stellar core to a neutron star assuming an axisymetry. The GW field amplitude emitted during the core collapse is derived from the quadrupole moment. By symetry (2D and axisymetry) there is only one non vanishing term $A_{lm}^{E2} = A_{20}^{E2}$. The GW amplitude is then given by

$$h_+ = \frac{1}{8}\sqrt{\frac{15}{\pi}} sin^2\theta \frac{A_{20}^{E2}}{R} \tag{2.25}$$

$$h_\times = 0$$

The ZM catalogue contain 78 waveforms obtained for different value of the following parameters:

- A: Angular momentum
- B: $\beta = \frac{E_{rot}}{E_{pot}}$ ratio between the rotational and the potential energies
- G: Γ_r reduced adiabatic indice

The Table 2.8.5 gives the initial values corresponding to the 78 ZM signals.

	A1	A2	A3	A4	A5
A [cm]	$5\ 10^9$	$1\ 10^8$	$5\ 10^7$	$1\ 10^7$	
	B1	B2	B3	B4	B5
β [%]	0.25	0.4	0.9	1.8	4.0
	G1	G2	G3	G4	G5
Γ_r	1.325	1.32	1.31	1.30	1.28

Table 2.1: Initial values of the parameters A, β and Γ_r used in the ZM simulation to produce the 78 waveforms of the catalogue

The time of bounce is defined as the moment when the central density reaches a (first) maximum. It mainly depends on the initial pressure reduction (parameter Γ_r) and can be described[9] as:

$$t_b(\Gamma_r) = (\frac{4}{3} - \Gamma_r)^{-1} t_1 + t_0$$

where $t_1 = 20.99$ ms and $t_0 = 0.63$ ms.

The class ZM provides facilities to get the GW amplitude corresponding to the 78 ZM simulated waveforms using equation (2.25) and assuming we are located in th equatorail plane ($\theta = \frac{\pi}{2}$).
The waveforms of the ZM catalogue [18] have been downsampled at 20 kHz.
The list of the 78 waveform file names is given in 2.17

[9]for all waveforms excecpt the one with β= 0.4 %, which are not considered in the 78 ZM waveforms

Include file	"BuS_ZM.hxx"
Constructors	ZM (string aName, double aSNR) ZM (string aName, int aDistance) ZM (string aName, double aSNR, int aSampling, Vector<double> &aPSD) ZM (string aName, int aDistance, int aSampling, Vector<double> &aPSD) ZM (string aName, double aSNR, ITF::Detector aITFName) ZM (string aName, int aDistance, ITF::Detector aITFName)
Public methods	int GetData (double aPosition, Vector<double>& aData) int GetData (double aPosition, Vector<float>& aData) int SetZM (string aName) int SetDistance (int aDistance)

The method *GetData()* fills the data vector with the chosen ZM signal which is located at the relative position given by the argument *aPosition*. This position is a fraction of the data vector size.
If the size of the ZM signal is longer than the vector data size a warning message is printed out and the vector is not filled up.

The method *SetZM(...)* changes the waveform by a new one.

The method *SetDistance(...)* initializes the distance and the normalization factor is recomputed.

Program example:

```
#include "BuS_ZM.hxx"
#include "DataContainer_Vector.hxx"
using namespace \_BuL;
int main(int arg, char* argc[])
{
  int i, j;
  int size = 2010;
  double position = 0.3;
  Vector<double> dou(size);

  ZM signal("a1b2g1", 10.);

  signal.GetData (position, dou);

  cout <<"Signal SNR=10" << endl;
  for (j=0 ; j<20 ; j++)
    cout << "Vector dou["<< j <<"]= " << dou[j] << endl;

  signal.SetDistance (10);
  signal.GetData (position, dou);

  signal.SetSNR (10.);
  signal.GetData (position, dou);

  cout <<"Signal SNR=10" << endl;
  for (j=0 ; j<20 ; j++)
    cout << "Vector dou["<< j <<"]= " << dou[j] << endl;
}
```

2.8.6 Class DFM

The Dimmelmeier & Font & Mueller catalogue gathers a collection of waveforms produced by the relativist simulation of the collapse of a rotating stellar core to a neutron star assuming an axisymetry. The GW field amplitude emitted during the core collapse is derived from the quadrupole moment. By symetry (2D and axisymetry) there is only one non vanishing term $A^{E2}_{lm} = A^{E2}_{20}$. The GW amplitude is then given by

$$h_+ = \frac{1}{8}\sqrt{\frac{15}{\pi}} sin^2\theta \frac{A^{E2}_{20}}{R} \tag{2.26}$$

$$h_\times = 0$$

The DFM catalogue contain 25 waveforms obtained for different value of the following parameters:

- A: Angular momentum
- B: $\beta = \frac{E_{rot}}{E_{pot}}$ ratio between the rotational and the potential energies
- G: Γ_r reduced adiabatic indice

The Table 2.8.6 gives the initial values corresponding to the 25 DFM signals.

	A1	A2	A3	A4	A5
A [cm]	5 10^9	1 10^8	5 10^7	1 10^7	
	B1	B2	B3	B4	B5
β [%]	0.25	0.5	0.9	1.8	4.0
	G1	G2	G3	G4	G5
Γ_r	1.325	1.32	1.31	1.30	1.28

Table 2.2: Initial values of the parameters A, β and Γ_r used in the DFM simulation to produce the 25 waveforms of the catalogue

The class DFM provides facilities to get the GW amplitude corresponding to the 25 DFM simulated waveforms using equation (2.26) and assuming we are located in th equatorail plane ($\theta = \frac{\pi}{2}$).
The waveforms of the DFM catalogue [20] have been downsampled at 20 kHz.

Include file "BuS_DFM.hxx"

Constructors

```
DFM (string aName, double aSNR)
DFM (string aName, int aDistance)
DFM (string aName, double aSNR, int aSampling, Vector<double> &aPSD)

DFM (string aName, int aDistance, int aSampling, Vector<double> &aPSD)
DFM (string aName, double aSNR, ITF::Detector aITFName)
DFM (string aName, int aDistance, ITF::Detector aITFName)
```

Public methods

```
int GetData (double aPosition, Vector<double>& aData)
int GetData (double aPosition, Vector<float>& aData)
int SetDFM (string aName)
int SetDistance (int aDistance)
```

The method *GetData()* fills the data vector with the chosen DFM signal which is located at the relative position given by the argument *aPosition.* This position is a fraction of the data vector size.
If the size of the DFM signal is longer than the vector data size a warning message is printed out and the vector is not filled up.

The method *SetDFM(...)* changes the waveform by a new one.

The method *SetDistance(...)* initializes the distance and the normalization factor is recomputed.

Program example:

```
#include "BuS_DFM.hxx"
#include "DataContainer_Vector.hxx"
using namespace \_BuL;
int main(int arg, char* argc[])
{
  int i, j;
  int size = 2010;
  double position = 0.3;
  Vector<double> dou(size);

  DFM signal ("a1b2g1", 10.);

  signal.GetData (position, dou);

  cout <<"Signal SNR=10" << endl;
  for (j=0 ; j<20 ; j++)
    cout << "Vector dou["<< j <<"]= " << dou[j] << endl;

  signal.SetDistance (10);
  signal.GetData (position, dou);

  signal.SetSNR (10.);
  signal.GetData (position, dou);

  cout <<"Signal SNR=10" << endl;
  for (j=0 ; j<20 ; j++)
    cout << "Vector dou["<< j <<"]= " << dou[j] << endl;
}
```

2.8.7 Class Signal

Include file	**BuS_Signal.hxx**

Constructors	`Signal ()`
	`Signal (Vector<double> &aPSD, int aSampling)`
	`Signal (double aSigmaNoise, int aSampling)`
	`Signal (ITF &aITF)`

Destructors	`~Signal ()`
Public methods	see below

```
int SetSignal (Vector<double> &aData)
int SetGaussianSignal (double aSigma)
int SetGaussianSineSignal (double aSigma, double aFrequency)
int SetGaussianCosineSignal (double aSigma, double aFrequency)
int SetDampedSineSignal (double aTau, double aFrequency)
int SetDampedCosineSignal (double aTau, double aFrequency)

double GetSNR ()
double GetHrss ()
double GetDistance ()

int SetSNR (double aSNR)
int SetHrss (double aHrss)
int SetDistance (int aDistance)

double GetSNRForHrss (double aHrss)
double GetHrssForSNR (double aSNR)

int SetPSD (int aSampling, Vector<double> &aPSD)
int SetITF (ITF &aITF)
int SetSigmaNoise (int aSampling, double aSigma)
double GetSigmaNoise ()
int GetSamplingFrequency ()
int GetData (double aPosition, Vector<double>& aData)
void SetDebug (bool aValue)
bool GetDebug ()
```

2.9 BuT: burst template generators

2.9.1 Class TGaussian

This class provides methods to define and generate templates of Gaussian form in time space as well as in Fourier space.
The Gaussian template wave form is given by

$$g(t) = e^{\frac{-(t-t_0)^2}{\sigma^2}} \tag{2.27}$$

where t_0 is chosen as half of the window size in such a way the template is centered.
By default, the Gaussian template is not normalized since it depends on a given PSD.

Include file "BuT_TGaussian.hxx"

Constructors TGaussian(double aSigmaMin, double aSigmaMax, double aEpsilon, int aSampling)
TGaussian(double aSigmaMin, double aSigmaMax, double aEpsilon, int aSampling, bool aPadding)

Public methods

```
int GetNTemplate()
double GetSigmaTemplate(int aIndex )
Vector<double>& GetSigmaTemplate(int aIndex)
void SetDebug(bool aValue)
bool GetDebug()
int GetWindowSize()
void SetWindowSize(int aValue)

int GenerateWaveForm()
int GenerateFourierWaveForm()

Matrix<double>& GetWaveForm()
Matrix<complex<double>>& GetFourierWaveForm()
```

When one creates a TGaussian object, the number of templates are computed. This number depends on the 3 parameters:

- σ_{min}: minimal value of the templates gaussian width
- σ_{max}: maximal value of the templates gaussian width
- ϵ: maximal value of the SNR loss. The minimal match (MM) is defined: MM=1-ϵ

The number of templates is given by:

$$N = \frac{log(\frac{\sigma_{max}}{\sigma_{min}})}{log(\frac{1+\gamma_+}{1+\gamma_-})} + 1$$

where γ_+ and γ_- are given by:

$$\gamma_\pm = \frac{\sqrt{1-MM^2}(\sqrt{1-MM^2} \pm \sqrt{1+MM^2})}{MM^2}$$

The different values of gaussian template width are determined using the recursive relation:

$$\sigma_{i+1} = \frac{1+\gamma_+}{1+\gamma_-}\sigma_i \qquad \sigma_0 = \sigma_{min}$$

The method *GenerateWaveForm()* fills a template matrix with values of Gaussian given in (2.27). The signal is centered at the middle of the data vector.

The method *GenerateFourierWaveForm()* fills a template matrix with Fourier Transform of the time domain Gaussian templates. One just applies a direct FFT on th etime domain wave form.

The methods *GetWaveForm()* and *GetFourierWaveForm()* return the pointer to the vector containing the template generated in time or frequency space.

One should mention that all templates are stored in vector of equal size given by the largest template σ_{max}.
The size of the vector containing the templates in time domain is the power of 2 just greater than $8 \times \sigma_{max} \times Sampling$ where $Sampling$ is supposed to be 20 kHz.
The size of the vector containing the frequency domain wave form is just identical or twice as big as the time domain wave form template size depending on the zero-padding option choice.

The method *SetWindowSize()* allows to re-set the value of the vector containing the time domain template wave form. This is interesting when the largest template is short is be able to set by hand the vector size of the template in order to have enought frequency resolution.

Finaly, one has to mention that by default the templates will be generated with zero padding. If one does not want to zero pad, one has to use the TGaussian constructor which allow to set-up the private zero padding variable to “false”.

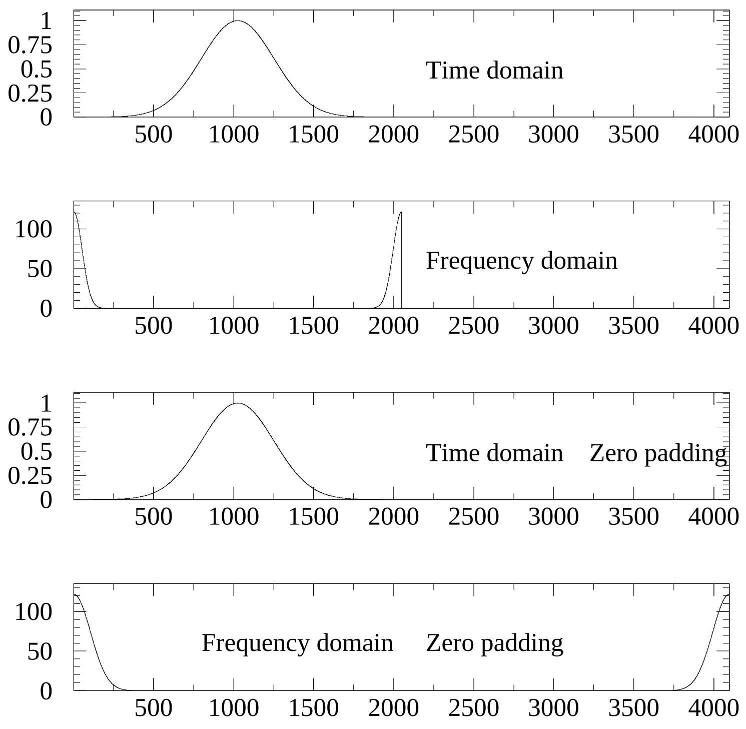

Figure 2.16: Exemple in time domain and in frequency domain of a Gaussian template wave form. One can see the different size of the vector depending on the use of zero-padding.

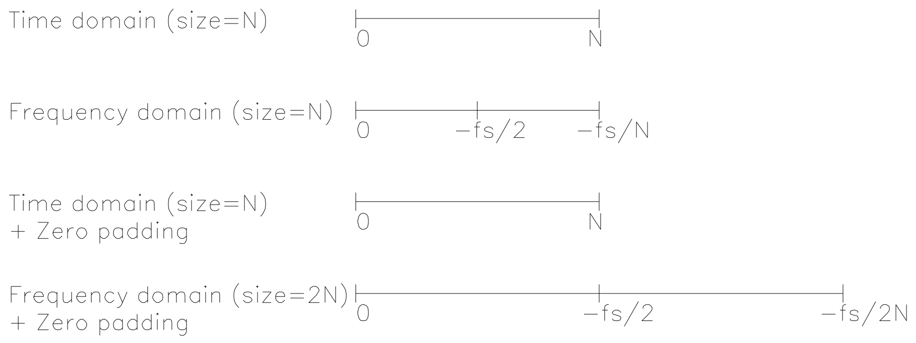

Figure 2.17: Representaion of the size of the time domain and frequency domain of the gaussian template

Program example:

```
#include <math.h>
#include "BuT_TGaussian.hxx"
#include <fstream>

using namespace \_BuL;

int main( int narg, char* argc[] )
{
  int i;

  double sigmaMin = 1.e-4;
  double sigmaMax = 1.2e-2;
  double epsilon = 1.e-2;
  int NTemplate;
  bool padding = true;
  int size;

  TGaussian TG(sigmaMin, sigmaMax, epsilon, padding);

  //  TG.SetDebug(true);
  TG.SetWindowSize(4096);

  NTemplate = TG.GetNTemplate();

  cout << "template #: " << NTemplate << endl;

  Vector<double> para;
  para = TG.GetSigmaTemplate();


  for (i=0 ; i<NTemplate ; i++)
    cout << "Template[" << i << "] = " << TG.GetSigmaTemplate(i)
         << " " << para[i] << endl;
  cout << "Template size " << TG.GetWindowSize() << endl;
  cout << "Generate WF " << endl;

  TG.GenerateWaveForm();
  cout << "Fill vector with WF " << endl;
  Matrix<double> WF;
  WF = TG.GetWaveForm();

  return( 0 );
}
```

2.9.2 Class TDampedSine

to be written ...

2.10 Tiler: Placement of templates in the parameter space

This package provides a tool to place template in a 2D parameter space.

2.10.1 Introduction

In order to insure a good signal recovery, the match filter technique often requires to run a large bank of templates in parallel. Then, the first problem to be solved is to design this bank in a suitable way. The usual prescription is to insure that the minimal match **MM** (minimum taken over the full bank) between any signal and the templates is above a value (0.97 is a typical choice).

2.10.1.1 Tiling formalism

As a linear filtering consists in correlating the detector output $s(t)$ with the corresponding filtering function $\varphi(t)$, one defines the following scalar product to represent the filtering operation:

$$\langle\, s \,|\, \varphi \,\rangle \;=\; 4\,\Re\left(\int_0^\infty df\, \frac{\tilde{s}(f)\,\tilde{\varphi}(f)^*}{S_h(f)}\right) \tag{2.28}$$

where S_h is the one-sided power spectrum density and the "~" symbol means Fourier transform. The normalization is chosen so that, in case of signal alone, $\sqrt{\langle s|s\rangle}$ is equal to the SNR.

The ambiguity function Γ between two templates $k_{\vec{\lambda}}$ et $k_{\vec{\lambda}+d\vec{\lambda}}$ is defined by:

$$\Gamma\left(\vec{\lambda}\,;\,d\vec{\lambda}\right) = \langle\, k_{\vec{\lambda}} \mid k_{\vec{\lambda}+d\vec{\lambda}} \,\rangle \tag{2.29}$$

The templates are normalized: $\langle\, k_{\vec{\lambda}} \,|\, k_{\vec{\lambda}} \,\rangle \;=\; 1 \;=\; \langle\, k_{\vec{\lambda}+d\vec{\lambda}} \,|\, k_{\vec{\lambda}+d\vec{\lambda}} \,\rangle$. Γ is thus a measurement of the closeness between two templates. It can also considered as a way to see how well the template $k_{\vec{\lambda}}$ can be used to detect the signal $k_{\vec{\lambda}+d\vec{\lambda}}$: the ambiguity function is the mean fraction of the optimal SNR achieved at a given distance in parameter space.

If $d\vec{\lambda}$ is small enough, the ambiguity function can be approximated by a second order power expansion:

$$\Gamma\left(\vec{\lambda}\,;\,d\vec{\lambda}\right) \;=\; 1 \;-\; \frac{1}{2}\,g_{\mu\nu}\,d\lambda^\mu\,d\lambda^\nu \tag{2.30}$$

with $g_{\mu\nu}$ defining a metric on the parameter space $\mathbb{P}$.

Finally, one defines the minimal match MM as the lower bound of the recovered SNRs, which means that the loss of the SNR due to the mismatch between the template and the signal must be kept below $1 - MM$. One can note that with this definition, a – of course *pessimistic* – estimator of the fraction of false dismissals $\mathfrak{L}$ is:

$$\mathfrak{L} \;=\; 1 \;-\; MM^3 \tag{2.31}$$

For instance, with $MM = 97\%$, one has $\mathfrak{L} \sim 10\%$. This value of MM is usually found in the literature as the correspondence with this particular value of $\mathfrak{L}$ is easy.

This quantity MM is the only input parameter of the tiling procedure; it allows one to define precisely the efficiency area $\mathbb{E}(\vec{\lambda}, MM)$ of the template $k_{\vec{\lambda}}$ by the following equation:

$$\frac{1}{2}\,g_{\mu\nu}\,d\lambda^\mu\,d\lambda^\nu \;\leq\; 1 \;-\; MM \tag{2.32}$$

The area of $\mathbb{P}$ including all the vectors $k_{\vec{\lambda}+d\vec{\lambda}}$ which match this inequality is the inner part of an ellipsoid centered on $\vec{\lambda}$ whose proper volume scales as $(1 - MM)^{-D_{\mathbb{P}}/2}$. The average fraction of recovered SNR for physical signals with parameters belonging to $\mathbb{E}(\vec{\lambda}, MM)$ is at least equal to MM.

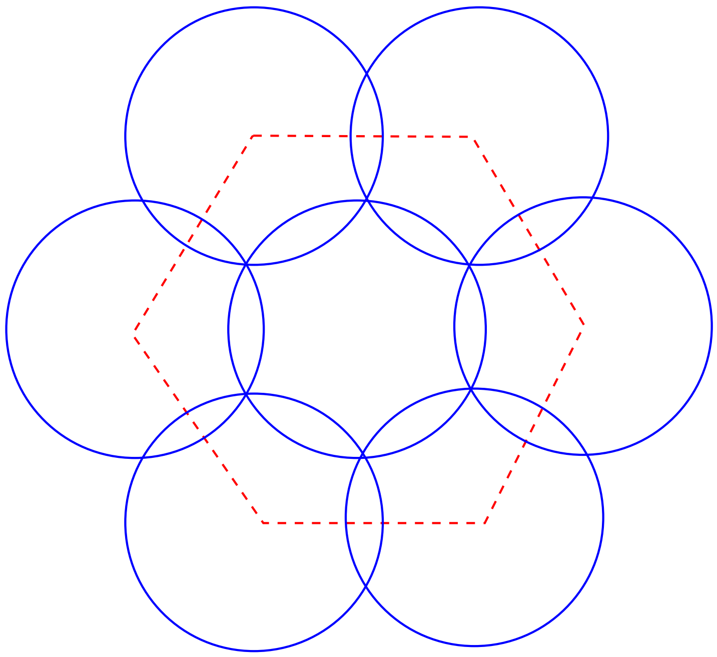

Figure 2.18: Optimal tiling of an infinite plane by identical circles: the centers belong to an hexagonal lattice.

2.10.1.2 The tiling problem

Tiling the parameter space consists of finding a set of $\mathfrak{N}$ filters $\left(k_{\vec{\lambda_p}}\right)_{1\leq p\leq\mathfrak{N}}$ so that the union of the surfaces $\left(\mathbb{E}(\vec{\lambda_p}, MM)\right)_{1\leq p\leq\mathfrak{N}}$ completely covers $\mathbb{P}$. The aim is to achieve this task by using as few templates as possible to keep the computing cost manageable.

The main problems encountered by any tiling procedure are: overlapping, gaps, areas lost beyond the physical parameter space... The difference between the ellipsoid and the real border of the efficiency area is not really taken into account: in practical cases for one-step searches, MM is close to 1 and thus the ellipsoid is assumed to be a good approximation of the ambiguity surface. Moreover, the overlapping between close ellipses is an advantage to discard such a problem. Clearly, the value of MM from which the ellipsoid approximation of the efficiency area is no longer valid depends on the precise tiling problem considered.

2.10.1.3 Tiler Algorithm

It is well-known that the optimal tiling of an infinite plane by identical disks of radius R consists in placing their centers on an hexagonal lattice, separated by a distance of $\sqrt{3}R$ – see Figure 2.18. In this way, the overlapping is minimized.

This property of circular tiling is used by the algorithm on the following way. Once the location of a template $\vec{k}_0$ in $\mathbb{P}$ has been chosen, one computes its efficiency area $\mathbb{E}_0$. Through a simple plane transformation, $\mathbb{E}_0$ becomes a circle $\mathbb{C}_0$ of unit radius and the six neighbor centers are placed on the regular hexagonal lattice previously described. This choice is nearly optimal if the shape of the efficiency areas is slowly varying with respect to their characteristic dimensions. Using the inverse transformation, the six new center positions are given in $\mathbb{P}$.

Then, each ellipse associated with a new center is tested in order to check if it must be kept. The conditions are that the center belongs to the parameter space and it does not fall in an ellipse already

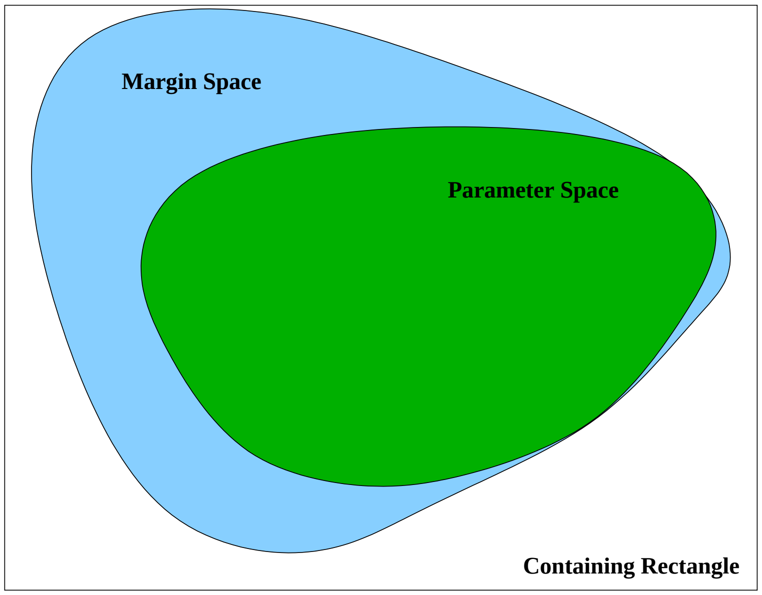

Figure 2.19: Definitions of the parameter space, the margin space and the containing rectangle.

in the list. It is also checked that the ellipse covers a part of $\mathbb{P}$ not already covered by other ellipses. The minimal surface to be covered is 0 by default and can be changed by the *SetMinimalCoveredSurface* method. This test is made doing a random generation of points in the ellipse: the number of tests can be defined by the *SetNbLoopsComputeCoveredSurface* method. By default, 10000 tests are performed.

If the ellipse is useful, its center is added to the center list and will be used later to place other centers. This iterative algorithm stops either when the full surface of the parameter space is covered or when no new ellipse can be placed any longer.

A second iteration is performed to treat the edges of the parameter space. The same procedure of placement is followed but centers outside the parameter space but inside the margin space (see Figure 2.19) are accepted.

Concerning the placement of the six neighbors, three parameters can be adjusted by the user: the angle offset θ_0, the turn direction $TD = \pm 1$ and the safety factor SF. In the unit circle referential, the nth neighbor is placed in polar coordinates at : $(SF, \theta_0 + TD * n * \pi/3)$. By default $\theta_0 = 0$, $TD = 1$ and $SF = 1$. Theses values can be set by the user using the corresponding methods (see TilerWithEllipses section).

2.10.2 Class TilerPoint

A simple class to define points in a n-dimensional space. It is used by other Tiler classes to define center of patterns or to test the parameter space.

Include file	**TilerPoint.hxx**

Constructors	`TilerPoint ( int aDim = 2 )`
	`TilerPoint ( const TilerPoint& aPoint )`

Destructors	`~TilerPoint ()`

Public methods	`double GetCoord ( int aN )`
	`int SetCoord ( int aN, double aValue )`
	`int GetDim ()`
	`static int GetMaxDim ()`
	`double Distance ( TilerPoint& aPoint )`

The constructor *TilerPoint (int aDim = 2)* creates a point in a *aDim* dimension space (by default set to 2) and set it at the origin.
The constructor *TilerPoint (const TilerPoint& aPoint)* creates a point which is an exact copy of *aPoint* (dimension and coordinates).

The *GetCoord(int aN)* method returns the aN^{th} coordinate. It returns TilerPoint::PointError if *aN* is out of range.

The *SetCoord(int aN, double aValue)* method sets the aN^{th} coordinate to *aValue*.

The *GetDim()* method returns the dimension of the space.

The *GetMaxDim()* method returns the maximal allowed dimension for the space.

The *Distance (TilerPoint& aPoint)* method computes the euclidian distance between the current point and *aPoint*.

2.10.3 Class TilerPattern

This is a generic class internally used by Tiler for the definition of geometrical patterns. No use is foreseen for standard users of the Tiler package.

Include file	**TilerPattern.hxx**
Constructors	`TilerPattern ( int aDim = 2 )`
Destructors	`~TilerPattern ()`

Public methods	`void SetCenter ( TilerPoint& aPoint )`
	`TilerPoint& GetCenter ()`
	`bool Characteristic ( TilerPoint& aPoint )`
	`void SetCharact ( bool (*aFunction) ( TilerPoint& aPoint ))`
	`double Area ()`

The constructor *TilerPattern (int aDim = 2)* creates a default pattern in a *aDim* dimension space (by default set to 2). The default pattern is an hypercube of side equal to 1 centered on the origin.

The *SetCenter (TilerPoint& aPoint)* method allows to define the center of the pattern. The center is set to *aPoint*.

The *GetCenter ()* method returns the center of the pattern.

The *Characteristic (TilerPoint& aPoint)* method allows to test if *aPoint* belongs or not to the pattern. This function returns *true* is aPoint belongs to the pattern and *false* if not.

The *SetCharact (bool (*aFunction) (TilerPoint& aPoint))* method allows to define the characteristic function of the pattern.

The *Area ()* method returns the area of the pattern.

2.10.4 Class TilerEllipse

This class inherits from TilerPattern. It defines the elliptic pattern. As TilerPattern, it is only for internal use of the Tiler package. No use should be done by standard users.

Include file	**TilerEllipse.hxx**
Constructors	`TilerEllipse ( double aX2Coeff = 1, double aY2Coeff = 1, double aXYCoeef = 0 )`
	`TilerEllipse ( const TilerEllipse& aEllipse )`
Destructors	`~TilerEllipse ()`

Public methods	`void SetX2Coeff ( double aValue )`
	`void SetY2Coeff ( double aValue )`
	`void SetXYCoeff ( double aValue )`

	`double GetX2Coeff ()`
	`double GetY2Coeff ()`
	`double GetXYCoeff ()`
	`double GetLongAxis ()`
	`double GetSmallAxis ()`
	`double GetTiltAngle ()`
	`bool Characteristic ( TilerPoint& aPoint )`
	`double Area ()`

An ellipse $\mathcal{E}$ centered on (x_0, y_0) is defined in the following way:

$$\mathcal{E} = \{(x,y) \in \mathbb{R}^2 \quad | \quad X2Coeff * (x - x_0)^2 + Y2Coeff * (y - y_0)^2 + 2 * XYCoeff * (x - x_0) * (y - y_0) \leq 1\}$$

Be careful with the factor 2 in the xy coefficient.

The constructor *TilerEllipse (double aX2Coeff = 1, double aY2Coeff = 1, double aXYCoeef = 0)* creates an ellipse centered on the origin with $X2Coeff = aX2Coeff, Y2Coeff = aY2Coeff$ and $XYcoeff = aXYCoeef$. By default, an unit circle is created.
The constructor *TilerEllipse (const TilerEllipse& aEllipse)* creates an ellipse which is an exact copy of *aEllipse* (center and ellipse coefficients).

The *SetX2Coeff (double aValue)* method sets X2Coeff to *aValue.*

The *SetY2Coeff (double aValue)* method sets Y2Coeff to *aValue.*

The *SetXYCoeff (double aValue)* method sets XYCoeff to *aValue.*

The *GetX2Coeff ()* method returns the X2 coefficient.

The *GetY2Coeff ()* method returns the Y2 coefficient.

The *GetXYCoeff ()* method returns the XY coefficient.

The *GetLongAxis()* method returns the value of the ellipse long axis.

The *GetSmallAxis()* method returns the value of the ellipse small axis.

The *GetTiltAngle()* method returns the value of the tilt angle between the x axis and the long axis.

The *Characteristic(TilerPoint& aPoint)* allows to test if *aPoint* belongs or not to the ellipse.

The *Area ()* returns the ellipse area.

2.10.5 Class TilerRectangle

This class inherits from TilerPattern. It defines the rectangular pattern. As TilerPattern, it is only for internal use of the Tiler package. No use should be done by standard users.

Include file	**TilerRectangle.hxx**
Constructors	`TilerRectangle ( double aLength = 1, double aWidth = 1, double aTilt = 0`
	`TilerRectangle ( const TilerRectangle& aRectangle )`
Destructors	`~TilerRectangle ()`

Public methods	`void SetLength ( double aValue )`
	`void SetWidth ( double aValue )`
	`void SetTilt ( double aValue )`

	`double GetLength ()`
	`double GetWidth ()`
	`double GetTiltAngle ()`
	`bool Characteristic ( TilerPoint& aPoint )`
	`double Area ()`

A rectangle is defined by its Center, its Length, its Width and its Tilt Angle defined as the agnle between the x axis and a large side.
The constructor *TilerRectangle (double aLength = 1, double aWidth = 1, double aTilt = 0)* creates an rectangle centered on the origin with $Length = aX2Coeff, Width = aY2Coeff$ and $Tilt = aTilt$. By default, an unit square is created.
The constructor *TilerRectangle (const TilerRectangle& aRectangle)* creates an rectangle which is an exact copy of *aRectangle* (center and rectangle coefficients).

The *SetLength (double aValue)* method sets the Length to *aValue*.

The *SetWidth (double aValue)* method sets the Width to *aValue*.

The *SetTiltAngle (double aValue)* method sets the Tilt Angle to *aValue*.

The *GetLength ()* method returns the Length.

The *GetWidth ()* method returns the Width.

The *GetTiltAngle()* method returns the value of the tilt angle between the x axis and a large side.

The *Characteristic(TilerPoint& aPoint)* allows to test if *aPoint* belongs or not to the rectangle.

The *Area ()* returns the rectangle area.

2.10.6 Class Tiler

This class a virtual one which can not be instancied. It will be inherited by all tiling procedures.

Include file **Tiler.hxx**

Constructors `Tiler( int aDim = 2 )`

Destructors `~Tiler ()`

Public methods

```
virtual int Tile() = 0
int GetDim()
void SetPrintPeriod( int aPeriod )
int GetPrintPeriod()
void SetDecreaseSurfaceFactor( double aValue )
double GetDecreaseSurfaceFactor()
void SetDebugLevel( int aValue )
int GetDebugLevel()

void SetBoundMin(int aIndex, double aValue)
double GetBoundMin( int aIndex )
void SetBoundMax(int aIndex, double aValue)
double GetBoundMax( int aIndex )

void SetCharacteristicFunction( bool (*aFunction)( TilerPoint& aPoint ) )

void SetMarginCharacteristicFunction( bool (*aFunction)( TilerPoint& aPoint ) )

void SetAreaFunction( double (*aFunction)() )
void SetSpaceParamArea( double aValue )
double GetSpaceParamArea()
double ComputeSpaceParamArea( double aAccuracy )

void SetStartingCenter( double* aCoord )
TilerPoint& GetStartingCenter()

void CentersToFile( std::string aFileName )
```

The constructor *Tiler(int aDim = 2)* creates a Tiler object for a parameter space of dimension *aDim*. By default, it works in a 2D space.

The virtual *Tile()* method is the one which will do the tiling for each derivated classes.

The *GetDim()* method returns the dimension of the parameter space.

For debug prints, the *SetPrintPeriod(int aPeriod)* method sets the period of print in the choose of centers. The program will print the following lines each time *aPeriod* centers have been tested in the tiling procedure:

```
6000 Centers chosen
Total Time Elapsed: 137 Since Previous: 14
```

It also gives the time elapsed since the start of the program and the time since the previous print. By default, the PrintPeriod is set to 1000.

The *GetPrintPeriod()* method returns the Print Period defined above.

The *SetDecreaseSurfaceFactor(double aValue)* method set the DecreaseSurfaceFactor variable. Each time, the remaining surface to be tiled divided by the total parameter space surface is decreased by this factor, the following print is done:

```
Remaining Surface = 2.07861 Used Ellipses = 359
Total Time Elapsed: 18 Since Previous: 6
```

The print gives the remaining surface to tile and the number of ellipses already placed It also shows the time elapsed since the start of the program and the time since the previous print. By default, the DecreaseSurfaceFactor is set to 10 %.

The *SetDebugLevel(int aValue)* method sets the debug level. Higher is the level, the more you get information. By default, the debug level is one. The maximal value is 3.

The *GetDebugLevel()* method returns the debug level.

The *SetBoundMin(int aIndex, double aValue)* method sets the *aIndex* minimal bound of the rectangle containing the parameter space to *aValue*.

The *GetBoundMin(int aIndex)* method returns the *aIndex* minimal bound of the rectangle containing the parameter space.

The *SetBoundMax(int aIndex, double aValue)* method sets the *aIndex* maximal bound of the rectangle containing the parameter space to *aValue*.

The *GetBoundMax(int aIndex)* method returns the *aIndex* maximal bound of the rectangle containing the parameter space.

The *SetCharacteristicFunction(bool (*aFunction)(TilerPoint& aPoint))* method sets the charateristic function of the parameter space. This function should return *true* or *false* if the point *aPoint* belongs or not to the parameter space.

The *SetMarginCharacteristicFunction(bool (*aFunction)(TilerPoint& aPoint)* method sets the charateristic function of the margin around the parameter space. This function should return *true* or *false* if the point *aPoint* belongs or not to the margin.

The *SetStartingCenter(double* aCoord)* method allows to define the coordinate of the starting point of the tiling procedure. The *aCoord* array is used to fill the starting point coordinates.

The *GetStartingCenter()* method returns the starting point of the tiling procedure.

2.10.7 Class TilerWithEllipses

Include file **TilerWithEllipses.hxx**

Constructors

```
TilerWithEllipses ()
```

Destructors

```
~TilerWithEllipses ()
```

Public methods

```
void SetX2Function ( double (*aFunc)(TilerPoint& aPoint) )
void SetY2Function ( double (*aFunc)(TilerPoint& aPoint) )
void SetXYFunction ( double (*aFunc)(TilerPoint& aPoint) )
void SetSafetyFactor ( double aValue )
double GetSafetyFactor ()
void SetAngleOffset ( double aValue )
double GetAngleOffset ()
void SetTurnDirection ( double aValue )
double GetTurnDirection ()
void SetMinimalCoveredSurface ( double aValue )
double GetMinimalCoveredSurface ()
void SetNbLoopsComputeCoveredSurface ( int aValue )
int GetNbLoopsComputeCoveredSurface ()

int Tile()
double TotalEllipsesArea ()
double EstimatedCoveredArea ()

void WriteKumac ( std::string aFileName, int aPeriod )
```

The constructor *TilerWithEllipses ()* creates the object. No parameter is needed.

The *SetX2Function (double (*aFunc)(TilerPoint&) aPoint)* method allows to define the function which returns the value of the X2Coeff for the ellipse centered on *aPoint*.

The *SetY2Function (double (*aFunc)(TilerPoint&) aPoint)* method allows to define the function which returns the value of the Y2Coeff for the ellipse centered on *aPoint*.

The *SetXYFunction (double (*aFunc)(TilerPoint&) aPoint)* method allows to define the function which returns the value of the XYCoeff for the ellipse centered on *aPoint*.

The *SetSafetyFactor (double aValue)* method sets the safety factor to *aValue*.

The *GetSafetyFactor ()* method returns the value of the safety factor.

The *SetAngleOffset (double aValue)* method sets the angle factor to *aValue*.

The *GetAngleOffset ()* method returns the value of the angle offset.

The *SetTurnDirection (double aValue)* method sets the turn direction. If *aValue is greater than zero, TurnDirection is anticlockwise and clockwise if not.*

The GetTurnDirection () method returns the turn direction. 1 means anticlockwise and -1 clockwise.

The SetMinimalCoveredSurface (double aValue) method sets the minimal covered surface criteria to aValue.

The GetMinimalCoveredSurface () method returns the value of the minimal covered surface criteria.

The SetNbLoopsComputeCoveredSurface (int aValue) method sets the number loops used to compute the surface covered by an ellipse in the parameter space to aValue.

The GetNbLoopsComputeCoveredSurface () method returns the previously defined variable.
The Tile() method performs the tiling of the parameter space. It returns the number of used ellipses.

The TotalEllipsesArea () method returns the sum of all ellipse areas. It should be call after the tiling.

The EstimatedCoveredArea () method returns the parameter space area covered by the ellipses.

The WriteKumac (std::string aFileName, int aPeriod) method writes a kumac file named aFileName for PAW. The following lines show an example of the kumac contents:

```
set faci 37
ellipse 3.71877 0.918781 0.0114446 0.301818 ! ! 51.906
set faci 38
ellipse 3.731 0.934381 0.0114497 0.301136 ! ! 51.908
wait
set faci 39
ellipse 4.08118 0.64727 0.011079 0.312601 ! ! 52.6857
set faci 40
ellipse 3.74324 0.949989 0.0114541 0.300423 ! ! 51.9139
```

The period of the "wait" command is given by aPeriod.

2.10.7.1 A TilerWithEllipses example

The Tiler package contains two examples: test_Tiler and Test_Triangle. The first one tiles a rectangle of given dimensions with unit circles. All circle centers must belong to the rectangle. The second test program tiles a squared triangle ABC, squared in B with A = (2,2); B = (2,4); C = (6,4) using circles of radius 0.25. No circle center can be put under the AC line. Here is an example of the output of the program and Figure 2.20 shows the result in the parameter space.

```
csh> test_Triangle.exe 3 3 1
Start Tiling procedure
Parameter space Area = 4
Total Time Elapsed: 0 Since Previous: 0
Remaining Surface = 3.80365 Used Ellipses = 1
Total Time Elapsed: 0 Since Previous: 0
Remaining Surface = 3.44484 Used Ellipses = 3
Total Time Elapsed: 0 Since Previous: 0
Remaining Surface = 3.09738 Used Ellipses = 5
Total Time Elapsed: 0 Since Previous: 0
Remaining Surface = 2.74301 Used Ellipses = 8
Total Time Elapsed: 0 Since Previous: 0
Remaining Surface = 2.24446 Used Ellipses = 11
Total Time Elapsed: 0 Since Previous: 0
Remaining Surface = 1.90807 Used Ellipses = 13
```

```
Total Time Elapsed: 0 Since Previous: 0
Remaining Surface = 1.43244 Used Ellipses = 16
Total Time Elapsed: 0 Since Previous: 0
Remaining Surface = 1.13078 Used Ellipses = 18
Total Time Elapsed: 0 Since Previous: 0
Remaining Surface = 0.702761 Used Ellipses = 21
Total Time Elapsed: 0 Since Previous: 0
Remaining Surface = 0.371225 Used Ellipses = 23
Before treating edges, surface not covered = 0.371225/4=> 9.28062 %
Before treating edges, Number of ellipses: 23
End of Tiling process, Surface not covered = 0.142635/4=> 3.56587 %
Number of ellipses: 35

Nb ellipses = 35
Space parameter area = 4
Area covered by the ellipses in the parameters space= 3.85737
Ratio = 96.4341 %
Total area covered by the ellipses = 6.87223
Ratio = 171.806 %
Write kumac file
Write kumac file Done
Centers List To File center.txt
Write Centers List  file Done
```

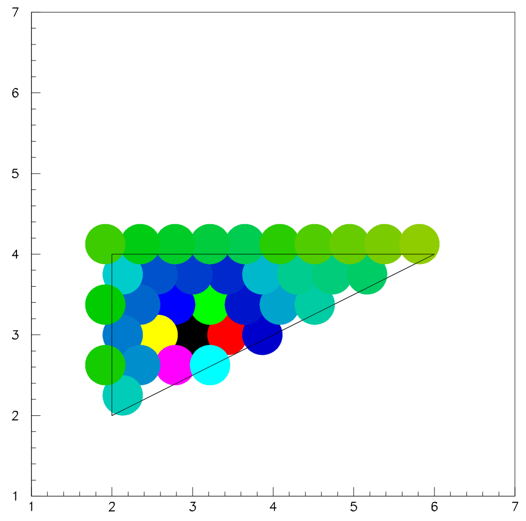

Figure 2.20: Tiling of a triangle by circles

2.11 BuF: burst filters

This package contains a Detector class which is a virtual class that provides the different methods which allow to define and apply the Burst algorithms to look for events hiden in chunk of data. Indeed, for all the Burst algorithms, it is mandatory first to define the input parameters (number of windows or templates, values of the threshold, ...) then to calculate the amplitude of the filters, to apply the thresholds on the amplitude and finally to cluster the different triggers output to define the events.
All the Burst algorithms classes which are described in the next sections derives from the class Detector and implements the virtual methods of the class Detector.
Each class algorithms is provided with a test application.

In addition to this, it contains a class DataCard whose role is to provide methods to read easily a data card.

Besides, we have given a simple definition of an "event" in the search of Burst context which leads to the definition of a class called Event.

2.11.1 How to treat the "end effects" between two data chunks

Since we have to break the data stream up into data chunks, the filter output will not be correct if one does not care about the fact that the beginning of each output is influenced by the last input data of the previous data chunks. Two technics are implemented to cope with "end effects" depending on the nature of the filters. Finally, it should be mention that the Burst filter classes offer the possibility to enable and disable the specific data chunk "end treatment". This is defined at the Configuration phase.

- *Time domain filters: ALF, NF and MF.*

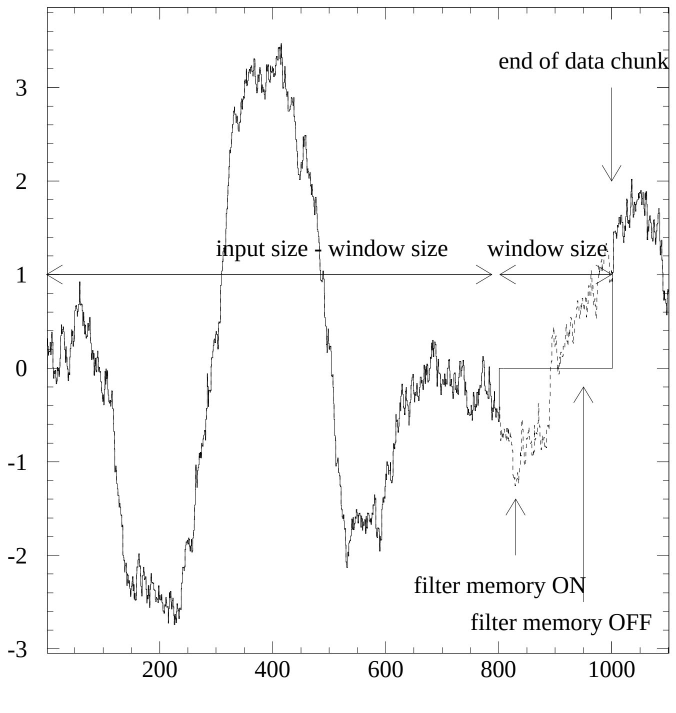

Figure 2.21: The full line histogram shows the output of the Mean Filter around a link between 2 data chunks without using the filter memory facility. The dashed histogram shows the same output but using the the filter memory facility.

A simple mecanism to save the filter memory has been implemented in the Detector class. It provides an alternative and efficient method to the data chunk overlapping that is necessary to be able to cover all the data stream. This filter memory mecanism consists in saving at each call of the Amplitude() (or *Detect()*) method the last part of the data chunk which are needed to be able to compute correctly the filter output of the next data chunk. This algorithm is derived from the one implemented in section 2.4.1.
The output of a filter at given time stamp t_i is computed using the input data located between t_{i-N} and t_i where N is the size of the window. Hence, for the first data chunk, the N points are not correct since they are computed with part of 0. One should also mention that this filter memory algorithm takes into account the fact that the filters are implemented with several windows in parallel.
Finally, it must be said that this mecanism imposes to work with constant input data vector size.

The comparison between the output around 2 data chunks junction of the MF filter using or not the filter memory option is shown in Figure 2.21.

- Correlator filters: PC and DSC We plan to implement the *overlap-and-add* nethod for Correlators (already used in Merlino). For the moment, the "end effect" is not treated. Hence it is recommended to overlap the data chunks.

2.11.2 Class Detector

The content of the class Detector is given for the sake of completion, but one can not instance such an object. There is only one public method which returns the name of the algorithm. The methods relative to the filter memory facility are protected.

Include file	*"BuF_Detector.hxx"*
Constructors	*Detector (string aName)*
Virtual methods	*bool Configure (string aFileName)* *bool Amplitude (Vector<double>& aInput, Matrix<double>& aAmplitude)* *int Threshold (Matrix<double>& aAmplitude, list<Event>& aEventList)* *bool Detect (Vector<double>& aInput, Matrix<double>& aAmplitude, list<Event>& aEventList)* *bool End ()*
Public method	*string GetName ()*
Non member function	*int Statistique (ofstream &aFile, int aGPS, list<Event> &aList)*

The functionalities of the 5 virtual methods may vary according to their specific implementation for each filter, but one can extract some common features:

By now, the algorithm configuration parameters are stored in an ASCII file whose name is passed as argument of the the method Configure(...). The syntax of this configuration file depends on the algorithm definition and hence on the specific implementation of the method *Configure(...)*.

The *Amplitude(...)* method computes for all the windows or templates the amplitude of the filter on the vector of data. It returs an array (windows/templates number $\times$ data vector size) containing the amplitude values.

The *Threshold(...)* method compares the threshold level defined (for each window or only once) to the amplitude vector. When the amplitude overcomes the threshold level, that defines the starting time (counted in bin) of a "micro-event". The ending time of this "micro-event" is defined as the next count for which the amplitude decreases below the threshold value. One can note that the previous definition of a starting time and an ending time for the micro-event has very little physical meaning in the case of the correlator filters. The physical width of the signal will rather be given by the width of the template which gives the highest output value in the case of several templates overcoming the threshold. All the other fields (see section 2.11.3) of the "micro-event" are defined according to obvious definition except the tolerance width which depends on the filter familly:

- case of the basic filters (NF, MF, ALF, ...)
  the tolerance width is defined as the size of the window which corresponds to this micro-event.
- case of the Peak Correlator filter (PC)
  the tolerance width is defined as twice the sigma value of the corresponding gaussian template.
- case of the Damped Sine Correlator filter (DSC)
  the tolerance width is defined by the value of the characteristic relaxing time of the template signal.

The *Threshold()* method provides at the end a (empty if not found) list of "micro-events" for all the windows.

It has been observed that when an true Burst event is present in the data, it often produces a set of "micro-events" for several which tend to cluster. In other word, these "micro-events" seem to correspond to the same physical "event" although this behavior is barely attented in the case of a false alarm (only one window trigger and of very short duration for example). If one clusters the "micro-events" one can reduce easily the number of false alarm by comparing the size of the events which are supposed to be larger in the case of a physical event compared to a false alarm.
Note that the clusterization of the "micro-events" will depend on the filters family (basic, correlator, ...) and will be performed by the user using the methods proposed in the class Event.

question en suspens .... doit on faire une classe Cluster ou bien les algos de clusterization sont ils des methods non membres de la classe Event ou Detector ....

The *Detect(...)* method provides a direct way to find out the events (the "micro-events" clusterization is also performed and produce a list of events) giving the vector of data as input argument.
Note that the *Detect(...)* method returns also the array of amplitude.

Finaly, the *End()* method is foreseen to provide some final information (print out or files/tuples filling for example) about the result of the algorithm applied on a chunk of data.

The *Statistique(...)* method writes in a file some information about the list of event.

2.11.3 Class Event

An event is defined by its:

- starting time (T_{start})
- ending time (T_{end})
- maximal amplitude value (A_{max})
- maximal amplitude time (T_{max})
- bin tolerance (σ)
- name of the algorithm which has defined the event
- type of the algorithm
- numero of template which has triggered the event
- number of "micro-events" which compose the event
- list of "micro-events" which have been merged in the event
- GPS time (approximation: 5.10^{-5} s)

All the quantities which concern time information are sampled in bin and are then relative to the first bin of the chunk of data which is analyzed.
The "bin tolerance" field is defined as the window size of the window which has triggered the event for basic filters or as twice the gaussian signal template width for the Peal Correlator filter. This information is used by the clusterization algorithm as it is explained in the following.
Note that when an event is the result of a clusterization of several "micro-events" the bin tolerance is redefined according to the clusterization algorithm as well as the numero of the template and the number of "micro-events" which compose the event is set to the right value.
Finally we keep an history of the "micro-events" which are stored in a list.

Include file	"BuF_Event.hxx"
Constructors	Event ()
Public methods	int GetTStart () int GetTEnd () int GetTMax () double GetAMax () int GetTolerance () string GetAlgoName () string GetAlgoType () int GetTemplateNumero () int GetTemplateNumber () double GetGPS () void SetTStart (int aValue) void SetTEnd (int aValue) void SetTMax (int aValue) void SetAMax (double aValue) void SetTolerance (int aValue void SetAlgoName (string aName) void SetAlgoType (string aName void SetTemplateNumero (int aValue) void SetTemplateNumber (int aValue) void SetGPS (double aValue) void PushBackMicroEvent (Event &aEvent) void PushFrontMicroEvent (Event &aEvent) void ClearMicroEvent () list<Event> GetMicroEvent () void SetDebug (bool aValue) bool GetDebug ()
Non member functions	ostream& operator << (ostream &aStream, Event aEvent) ostream& operator << (ostream &aStream, Event aEvent) void SortStart(list<Event>& aList) list<Event> ClusterBase(list<Event>& aList) list<Event> ClusterCorrelatorTight(list<Event>& aList) list<Event> ClusterCorrelatorLoose(list<Event>& aList)

Some non member functions have been defined:

- The operator << have been defined for an event and a list of event in order to print out in a pretty format an event or a list of events.

Program example:

```
list<Event> macroEvent = clusterBase (eventList);

cout << "Macro events found: " << endl;
cout << macroEvent << endl;
```

produces:

```
(Ts,Te) = (2446,2469)   Tmax: 2456   Amax: -7.18573  Template: -1 #T: 2  GPS: 0  Tolerance: 40  Micro-event template: 3, 5
(Ts,Te) = (2566,2625)   Tmax: 2571   Amax: 6.25865   Template: -1 #T: 3  GPS: 0  Tolerance: 40  Micro-event template: 0, 1, 4
(Ts,Te) = (2865,2865)   Tmax: 2865   Amax: -5.05144  Template: 0  #T: 1  GPS: 0  Tolerance: 10  Micro-event template: none
(Ts,Te) = (11041,11041) Tmax: 11041  Amax: 5.48995   Template: 0  #T: 1  GPS: 0  Tolerance: 10  Micro-event template: none
```

- The *SortStart(...)* methods sort in increasing start time order a list of events.

- The *ClusterBase(...)* clusterizes a list of events produced by a basic filter according to the following procedure:

- sort in the increasing start time order the list of event
- combine any ordered couple (i,j) of events if $T^i_{end} + \sigma^i > T^j_{start}$.
  - the 2 events are then combined to define a new event k.
  - $T^k_{start} = T^i_{start}$
  - $T^k_{end} = \max(T^i_{end}, T^j_{end})$
  - $A^k_{max} = \max(A^i_{max}, A^j_{max})$
  - T^k_{max} corresponds to A^k_{max}
  - $\sigma^k = \max(\sigma^i, \sigma^j)$
  - Template Numero = -1
  - Template Number += 1 - Micro-event list contains the 2 events
- remove the events i and j from the list of events
- add the event k in the list of events
- continue to cluster the (modified) list of events

- The *ClusterCorrelatorTight(...)* clusterizes a list of events produced by a correlator filter according to the following procedure:

- sort in the increasing start time order the list of event
- combine any ordered couple (i,j) of events if $T^i_{end} + \sigma^i > T^j_{start}$. The 2 events are then combined to define a new event k. Let's asume that i is the event index whose amplitude at maximum is the highest, then we have:
  - $T^k_{start} = T^i_{start}$
  - $T^k_{end} = T^i_{end}$
  - $A^k_{max} = A^i_{max}$
  - $T^k_{max} = T^i_{max}$
  - $\sigma^k = \sigma^i$
  - Template Numero = i
  - Template Number += 1 - Micro-event list contains the 2 events
- remove the events i and j from the list of events
- add the event k in the list of events
- continue to cluster the (modified) list of events

- The *ClusterCorrelatorLoose(...)* is identical to the *ClusterCorrelatorTight(...)* algorithm except that one defines and uses internally the end time of a macro event as the maximum of the 2 micro-events end time and the one of the maximum amplitude template. This enlarges the clusterization.

The *PushBackMicroEvent(...)* method does a push-back of an event into the list of micro-events.

The *PushFrontMicroEvent(...)* method does a push-front of an event into the list of micro-events.

The *ClearMicroEvent()* method clears the list of micro-events.

The *GetMicroEvent()* returns a copy of the list of micro-events.

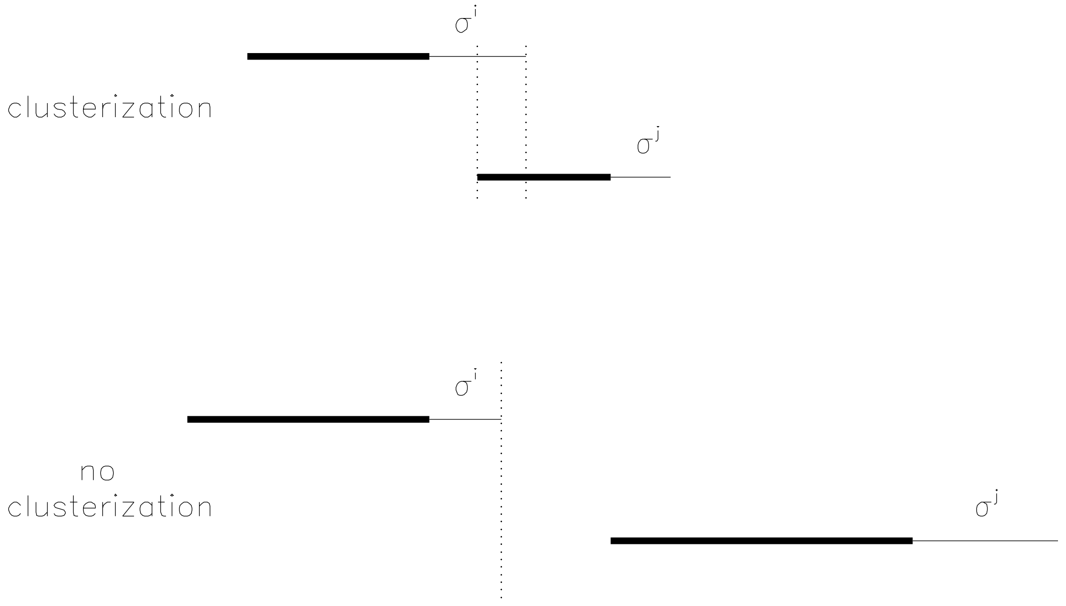

Figure 2.22: Clusterization of 2 “micro-events” (top) and example of 2 micro-events which do not fulfill the criteria to be clusterized (bottom).

2.11.4 Class Example or "how to code a new Burst algorithm"

This class is a skeleton class derived from Detector which can be taken as the starting point to define a new Burst algorithm class. All the virtual methods of the class Detector are implemented but are empty.

The test application source file is "test_BuF_Example.cxx".

Include file	"BuF_Example.hxx"
Constructors	Example(string aName)
Public methods	bool Configure(string aFileName) bool Amplitude(Vector<double>& aData, Matrix<double>& aAmplitude) int Threshold(Matrix<double>& aAmplitude, list<Event>& aEventList) bool Detect(Vector<double>& aData, Matrix<double>& aAmplitude, list<Event>& aEventList) int End() int GetNWindow() int GetWindow(int aIndex) void SetDebug(bool avalue) bool GetDebug()

The method *GetNWindow()* returns the number of window which will be used in paralell.

The method *GetWindow(int aIndex)* returns the size of the window which corresponds to the nimber *aIndex.*

"test_BuF_Example.cxx" program

```
#include <math.h>
#include "BuF_Example.hxx"
#include "DataContainer_Matrix.hxx"
#include "Noise_Gaussian_MT.hxx"
#include "BuS_Gaussian.hxx"

using namespace \_BuL;

int main( int narg, char* argc[] )
{
  int i;
  int size = 2048;

  Matrix<double> Amplitude( 10, size);
  Vector<double> h(size);
  Vector<double> g(size);
  list<Event> eventList;

  // Example created
  Example algo("Example");
  //algo.setDebug(true);

  //Generator created
  double sigmaNoise = 1.;
  Gaussian_MT Noise(sigmaNoise);
  Noise.GetData(h);

  //Source
  double sigma = 0.001;
  double SNR = 20.;
  double position = .5;

  Gaussian Signal(sigma, SNR);
  Signal.GetData(position, g);

  for (i=0; i<size ; i++)
    h[i] = h[i]+g[i];

  algo.Configure( "../test/Example.txt" );
  algo.Amplitude( h, Amplitude);

  int eventNumber = algo.Threshold( Amplitude, eventList);

  if (eventNumber > 0)
    {
      cout << "Micro events found :" << endl;
      cout << eventList << endl;

      // clusterization
      list<Event> macroEvent = ClusterBase( eventList );

      cout << "Macro events found: " << endl;
      cout << macroEvent << endl;
    }

// Alternative method
  algo.Detect( h, Amplitude, eventList);
  algo.End();


  return( 0 );
}
```

Here follows of an example of data card used for the configuration.

```
# ALF PARAMETERS DEFINITION
@ ALGO         MF
@ CLUSTER      Base
@ FILTERMEMORY No
#
@ SAMPLING 20000
@ NWINDOW   10
@ WINDOWS   10    15    20    25    30    40    50    70    150   200
@ THRESHOLD 5.33  5.33  5.33  5.33  5.33  5.33  5.33  5.33  5.33  5.33
```

2.11.5 Class ALF

The class ALF implements the algorithms described in [23]. The algorithm is coded in the public method *Regression(...)*. This function computes the linear regression of N points over a moving window of size n. The output result is returned in a vector filled up to bin (N-n-1). The end of the vector (from bin (N-n) up to bin (N-1)) is filled with zeros.

The filter output follows a Chi-Square probability distribution.
The filter output is given by:

$$ALF = \frac{X_a^2 + X_b^2 - 2\alpha X_a X_b}{1-\alpha^2}$$

$$\alpha = cov(X_a, X_b) = -\sqrt{\frac{3}{2}\frac{n+1}{2*n+1}}$$

where X_a and X_b are the normalised slope and the offset of the linear regression calculated in the moving window of size n.

The test application source file is "test_BuF_ALF.cxx".

Include file	"BuF_ALF.hxx"
Constructors	ALF(string aName)
Public methods	

```
int Configure (string aFileName)
int Configure (Vector<int>& aWindowSize, Vector<double>& aThreshold
                int aSampling)
int Amplitude (Vector<double>& aData, Matrix<double>& aAmplitude)
int Threshold (Matrix<double>& aAmplitude, list<Event>& aEventList )
int Threshold (Matrix<double>& aAmplitude, double &aGPS,
                list<Event>& aEventList )
int Detect (Vector<double>& aData, Matrix<double>& aAmplitude,
             list<Event>& aEventList)
int Detect (Vector<double>& aData, double &aGPS, list<Event>& aEventList )
int End ()
int SetThreshold (Vector<float> aThreshold)
int GetThreshold (Vector<float>& aThreshold)
int GetNWindow ()
int GetWindow (int aIndex)
void SetDebug (bool avalue)
bool GetDebug ()

int Regression (Vector<double>& aData, int aWindow, Vector<double>& aOutput)
```

The method *SetThreshold(...)* changes the initial values of the thresholds defined for each window.

The method *GetThreshold(...)* returns the value of the thresholds.

The method *GetNWindow()* returns the number of windows which are used in paralell.

The method *GetWindow(int aIndex)* returns the size of the window corresponding to the index *aIndex*.

2.11.6 Class MF

The class MF implements the algorithm described in [24]. The algorithm is coded in the public method *meanCompute(...)*. This function computes the mean of N points over a moving window of size n. The output result is returned in a vector filled up to bin (N-n-1). The end of the vector (from bin (N-n) up to bin (N-1)) is filled with zeros. The filter output follows a gaussian probability distribution.

The filter output is given by:

$$MF = \sum_{i=k}^{i=k+n} \frac{x_i}{\sqrt{n}}$$

Note that the output for each window is normalized and the threshold level may be the same for all the windows.

The test application source file is "test_BuF_MF.cxx".

Include file	"BuF_MF.hxx"
Constructors	MF(string aName)

Public methods

```
int Configure (string aFileName)
int Configure (Vector<int>& aWindowSize, Vector<double>& aThreshold,
                int aSampling)
int Amplitude (Vector<double>& aData, Matrix<double>& aAmplitude)
int Threshold (Matrix<double>& aAmplitude, list<Event>& aEventList )
int Threshold (Matrix<double>& aAmplitude, double &aGPS,
               list<Event>& aEventList )
int Detect (Vector<double>& aData, Matrix<double>& aAmplitude,
            list<Event>& aEventList)
int Detect (Vector<double>& aData, double &aGPS, list<Event>& aEventList )
int End ()
int SetThreshold (Vector<float> aThreshold)
int GetThreshold (Vector<float>& aThreshold)
int GetNWindow ()
int GetWindow (int aIndex)
void SetDebug (bool avalue)
bool GetDebug ()
int MeanCompute (Vector<double>& aData, int aWindow, Vector<double>& aOutput)
```

The method *SetThreshold(...)* changes the initial values of the thresholds defined for each window.

The method *GetThreshold(...)* returns the value of the thresholds.

The method *GetNWindow()* returns the number of windows which are used in paralell.

The method *GetWindow(int aIndex)* returns the size of the window corresponding to the index *aIndex*.

2.11.7 Class NF

The class NF implements the algorithms described in [22]. The algorithm is coded in the public method *normFilter(...).* This function computes the sum of the squared value of N points over a moving window of size n. The output result is returned in a vector filled up to bin (N-n-1). The end of the vector (from bin (N-n) up to bin (N-1)) is filled with zeros. The filter output follows a gaussian probability distribution when n is large enough (above 30).

The filter output is given by:

$$NF = \sum_{i=k}^{i=k+n} \frac{x_i^2}{\sqrt{2n-1}}$$

Note that the output for each window is normalized and the threshold level may be the same for all the windows.

The test application source file is "test_BuF_NF.cxx".

Include file	"BuF_NF.hxx"
Constructors	NF(string aName)

Public methods

```
int Configure (string aFileName)
int Configure (Vector<int>& aWindowSize, Vector<double>& aThreshold,
                    int aSampling)
int Amplitude (Vector<double>& aData, Matrix<double>& aAmplitude)
int Threshold (Matrix<double>& aAmplitude, list<Event>& aEventList )
int Threshold (Matrix<double>& aAmplitude, double &aGPS,
                    list<Event>& aEventList )
int Detect (Vector<double>& aData, Matrix<double>& aAmplitude,
                list<Event>& aEventList)
int Detect (Vector<double>& aData, double &aGPS, list<Event>& aEventList )
int End ()
int SetThreshold (Vector<float> aThreshold)
int GetThreshold (Vector<float>& aThreshold)
int GetNWindow ()
int GetWindow (int aIndex)
void SetDebug (bool avalue)
bool GetDebug ()
int NormFilter (Vector<double>& aData, int aWindow, Vector<double>& aOutput)
```

The method *SetThreshold(...)* changes the initial values of the thresholds defined for each window.

The method *GetThreshold(...)* returns the value of the thresholds.

The method *GetNWindow()* returns the number of windows which are used in paralell.

The method *GetWindow(int aIndex)* returns the size of the window corresponding to the index *aIndex.*

2.11.8 Class PC

The class PC implements the algorithm described in [22]. This a Wiener filtering using Gaussian form templates. The algorithm is coded in the public method *correlation(...)*. This function computes the correlation function between the data vector of size N and some gaussian templates of width σ_i which are generated and stored in memory at the configuration time. The output result is returned in a Matrix of size $TemplateNumber \times N$. The filter output follows a gaussian probability distribution.

Note that the output for each template is normalized and then the threshold level is identical for all the templates.

The test application source file is "test_BuF_PC.cxx".

Include file	"BuF_PC.hxx"
Constructors	PC(string aName)

Public methods

```
int Configure (string aFileName)
int Configure (Vector<double>& aSigma, double aThreshold, int aSampling,
              int aTemplateSize )
int Amplitude(Vector<double>& aData, Matrix<double>& aAmplitude)
int Threshold(Matrix<double>& aAmplitude, list<Event>& aEventList )
int Threshold (Matrix<double>& aAmplitude, double &aGPS,
              list<Event>& aEventList )
int Detect (Vector<double>& aData, Matrix<double>& aAmplitude,
           list<Event>& aEventList)
int Detect (Vector<double>& aData, double &aGPS, list<Event>& aEventList )
int End ()

void EnableRefreshPSD (bool aValue)
int UpdateCurrentPSD (Vector<double> aData)

int GetNTemplate ()
int GetWindowSize ()
double GetSigmaTemplate (int aIndex)

void SetDebug (bool aValue)
bool GetDebug ()
int GetSamplingFrequency ()
int SetSamplingFrequency (int aValue)

int Correlation (Vector<double>& aData, Matrix <double>& aAmplitude)
```

The gaussian templates definition is given in the PC filter datacard. The parameters are the minimal and maximal sigma values (σ_{max}, σ_{min}) as well as the maximal SNR loss (ϵ).
In order to reduce the wrap-around pollution when doing correlation using FFT, zero-padding is used internally[10]. Besides, to reduce the leakage effect when doing a FFT on a square data window, we have used an improved data windowing method. Indeed, if one simply convolves the data chunk with a moving taper window one finally loose information contained in the data located at the extremity of the window. To avoid such an effect we use the same method as the Welch Overlaping Approach usually used for Spectral estimation. That is to say, the correlation tapered window is moved with a stride equals to half

[10]The user does not care about adding zero-padiing at the end of the input data vector

of its size as shown in Figure 2.23 and the PC output amplitude is the combinaison (sum in time domain) of all the tapered window output. In that way, all points have identical weight in the amplitude output.

Finalyy, by default, the PSD is estimated using the input data each time the amplitude is computed. This default behavior can be changed at any moment, for instance when the input data are not good enough as just after a de-lock period. In that case the PC algorithm will used the previously estimated PSD.

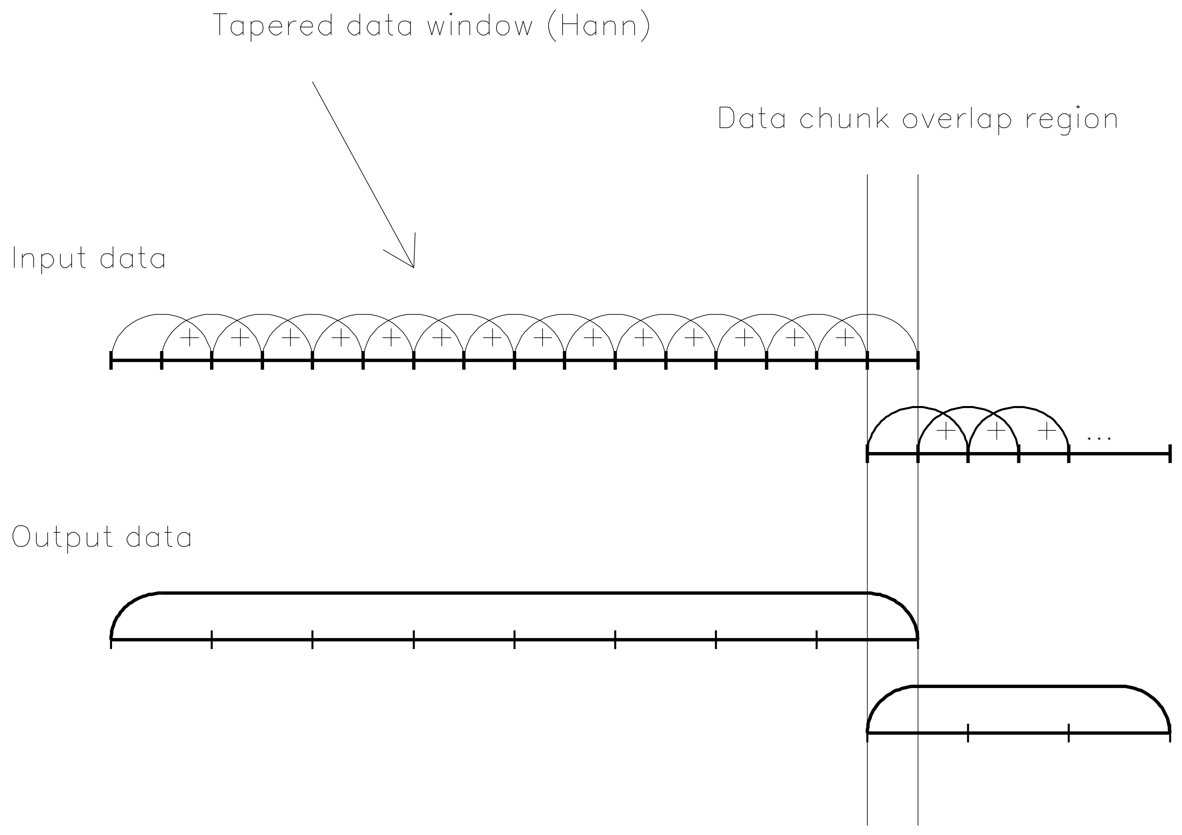

Figure 2.23: The Welch Overlapping Approach is used to compute the correlation between small data window and Gaussian template. This method has the advantage to give an identical weight to all the input data points, as shown in lower part of the figure.

An example of a PC datacard is given below:

```
# Example PARAMETERS DEFINITION
@ ALGO      Peak Correlator
@ CLUSTER   CorrelatorLoose
@ FILTERMEMORY No
#
@ SIGMA   1.0e-4 1.2e-2 // (en seconde)
@ EPSILON 1.0e-2
@ WINDOW 16384
@ THRESHOLD 5.
```

The *Configure(...)* method computes the number of templates and generates the wave forms in the

Fourier space which are stored in memory.

The *Amplitude(...)* method calculates for all the templates the correlation function between the templates and the data vector using the Fourier Transform of both. The filter output is given for template i using the Fourier Transfrom by:

$$PC_i(t) = \frac{1}{\sqrt{\pi f_s \sigma_i}} \mathcal{F}^{-1} \frac{(\mathcal{F}(\text{data})\mathcal{F}(\text{gauss}_{\sigma_i})^*)}{\text{PSD(f)}}$$

Note that the filter output is normalized in order to get a normal distribution (σ=1) in the hypothesis of a gaussian noise. Practically, the templates are normalized.

The *Threshold(...)* method compares the filter output vector to the threshold value and then produce a list of "micro-events" if to be. This list of micro-events can be clusterized if necessary to reduce the false alarm rate. It is recommended to use one of the algorithm devoted to Correlator filters *ClusterCorrelatorLoose(...)* or *ClusterCorrelatorTight(...)* methods to perform the event clusterization.

The method *EnableRefreshPSD(...)* method allows to disable or enable the estimation of the PSD which is done each time one calls the *Amplitude()* or *detect()* methods.

The method *UpdateCurrentPSD(...)* method allows to update the internally defined PSD at any moment giving a vector of data for the estimation.

The method *SetThreshold(...)* changes the initial values of the thresholds defined for each window.

The method *GetThreshold(...)* returns the value of the thresholds.

The method *GetNWindow()* returns the number of windows which are used in paralell.

The method *GetWindow(int aIndex)* returns the size of the window corresponding to the index *aIndex.*

2.11.9 Class DSC

to be written ...

2.11.10 Class PF

to be written ...

2.11.11 Class GDF

not yet available ...

2.12 TeF

2.12.1 Intro

Time-frequency transforms all propose to layout local signal frequencies. We can already distinguish between short time Fourier transform based analysis and wavelet (or multi-resolution).
When the Fourier transform is used, time resolution depends as usual on the delay between successive windows, but also on the window shape and other operations done on the signal. Frequency resolution depends on the window length, plus intermediate operations.
For wavelet transforms, especially a trous algorithm which is a shift invariant adaptation of the discrete wavelet transform, time resolution is fixed by the chosen wavelet, whereas frequency resolution is defined by the algorithm[11]. Most useful information can be found in [25] and [26].

The classes in this library is organized by analysis step I) signal transformation *Transforms* II) transform data mining *Image* III) detection algorithms (TBD)[12], plus several tool classes developed for a specific use but with a larger potential *Cluster* and *Histo*.

2.12.2 Class Transforms

Include file **TeF_Transforms.hxx**

Constructors

```
Transforms (const int aSignalFreq,const int aTailleIn, const bool aDebug)
```

Public methods

```
int Analytic (Vector<double>& aData)
int GaussianWindow (const int aTransformLength, const double aGaussianT0)

int HanningWindow (const int aTransformLength, const double aFactor)

int GaborParametrize (const int aTransformTLength, const int aTransformTStep)

int Gabor (Vector<double>& aData, Matrix<double>& aMap)

int AutoCorrParametrize (const int aTransformTLength, const int aTransformTStep)

int AutoCorr (Vector<double>& aData, Matrix<double>& aMap)

int PWVParametrize (const int aTransformTLength, const int aTransformTStep)

int PWV (Vector<double>& aData, Matrix<double>& aMap)
int PWVgauss (Vector<double>& aData, Matrix<double>& aMap)

int STransformSetEdge (const int aEdgeSize)
int STransformParametrize (const int aMinFreq, const int aMaxFreq, const double aTransfor

int STransform (Vector<double>& aData, Matrix<double>& aMap)
int STransform (Vector<double>& aData, Matrix<complex<double> >& aMap)

bool STransformProgressive (Vector<double>& aData, Vector<double>& aSTed)
```

[11] Only dyadic down-sampling is available yet.
[12] One of the reasons why frequency (to Hz) and time conversion (to GPS) are not implemented yet.

```
int STransform (Vector<double>& aData, Matrix<double>& aMapAbs, Matrix<double>& aMapAr

int STransformWelchParametrize (const int aMinFreq, const int aMaxFreq, const double aTra

int STransformNLSpacingWelchParametrize (const int aMinFreq, const int aMaxFreq, const do

int STransformASigmaWelchParametrize (const int aMinFreq, const int aMaxFreq, const doubl

int STransformWelch (Vector<double>& aData, Matrix<double>& aMap1, Matrix<double>& aMa


int ATrousParametrize (const int aMother)
int ATrous (Vector<double>& aData, Matrix<double>& aMap)

int DyadicParametrize (const int aMother)
int Dyadic (Vector<double>& aData, Matrix<double>& aMap)

void BuildFrequencyProfile (Matrix<double>& aMap, Vector<double>& aProfile)

void BuildFrequencyProfileAndSigma (Matrix<double>& aMap, Vector<double>& aProfile, Vec


int GetDim1 (void)
int GetDim2 (void)
int GetOverlap (void)
double GetAbsMax (void)

int GetDim1Offset (void)
int GetDim2Offset (void)

double GetDim1Min (void)
double GetDim1Step (void)
double GetDim2Min (void)
double GetDim2Step (void)

int SetStartTime (const double aTime)
int GetTimeLength ()
int GetFrequencyLength ()

double GetTimeMin (void)
double GetTimeStep (void)
double GetFrequencyMin (void)
double GetFrequencyStep (void)

void GetMapTimes (Vector<double>& aVector)
void GetMapFrequencies (Vector<double>& aVector)
void GetMapSigmas (Vector<double>& aVector)
double GetNLSpacingCoef (void)

void WriteTransformWindow (const string aFileName)
void WriteMapTimes (const string aFileName)
void WriteMapFrequencies (const string aFileName)
int WriteMapFile (const Matrix<double>& aMap, const int aMaxDim1, const int aMaxDim, con


void SetDebug (const bool aValue)
```

```
bool GetDebug ()
```

Each transform method comes with a parameterizing method, that sets up the required objects and allocates memory. It is mandatory to call the parameterizing method before using any method. Repetitive calls of a transform on chunks of data of the same size are possible.

Analytic computes the analytic signal related to the input data. It is the inverse FFT of the data FFT, after setting all negative frequencies coefficients to zero. It is used in the PWV transform, to reduce border effects in frequency.

When a transform relies on a specific time window shape, it is defined by calling either *GaussianWindow* for Gaussian profiles of sigma *T0* (in seconds) centered on the middle of the window, or *HanningWindow* of the shape

$$\alpha + (1-\alpha) * \cos(2 * \pi * \frac{k}{N})$$

for other purposes. The window coefficients are not affected when calling the transform, therefore redefinition is required only to change their parameters.

The standard parameters *aTransformTLength* and *aTransformTStep* set the size (in samples) of the window applied to the data and the number of samples between consecutive computations of the transform.

Gabor computes the square of the FFT of windowed data of the set length, with the previously defined window profile.

$$gabor(t,\nu) = \int_{-\infty}^{+\infty} e^{-2\pi i\nu\tau} x(t+\tau)\, d\tau$$

A trade-off can be achieved between *aTransformTStep* and the redundancy of the transform and the narrowness of the Gaussian profile. The limit for a flat profile is a spectrogram, for which the delay between two computations should be the length of the window.

PWV computes the pseudo Wigner Ville transform of the windowed data. It is the windowed version of the Wigner Ville transform.

$$WV(t,\nu) = \int_{-\infty}^{+\infty} e^{-2\pi i\nu\tau} x(t+\tau) * x^*(t-\tau)\, d\tau$$

$$PWV(t,\nu) = \sum_{k=-\frac{N}{2}}^{+\frac{N}{2}} e^{-2\pi i\nu k} x(t+k) * x^*(t-k)$$

The only way to obtain a high time resolution is to have

$$aTransformTStep = 1$$

thus incurring a high computational cost. It does not seem a recursive algorithm would be possible since the product between data elements are all different from a transform to another. For a given transform, it is this folding scheme that provides the time resolution, whereas frequency resolution relies on window length.

PWVgauss is personal test of a probably useless modified version of *PWV* where the window has a Gaussian profile, so as to reduce the terms away from the window centre.

ST... regroups all work done on the S transform and variations, internal Welch windowing and non linear template spacing.

ATrous is a shift invariant version[29] of *Dyadic* , which is the standard discrete wavelet transform (DWT), though computed using a lifting scheme[30]. Both use a dyadic scale, meaning the distance between correlated samples is always a power of 2. Starting at the sampling frequency, the frequency range is therefore not regularly estimated.
Better resolution would be possible using the continuous version of the wavelet transform, adapted to discrete data.
The maximum scale is

$$S_{max} = \log_2(datalength) + 1$$

making it necessary to use a mirror condition at both ends of the data block, to provide values for all wavelet coefficients (instead of a constant, maybe zero, value).
There is only one wavelet available at the moment for the a trous transform, the cubic b-spline, whereas Haar and so-called 9-7 wavelets are also implemented for the DWT.

Future additions :

- Continuous wavelet transform, to have a better frequency resolution, with an algorithm close to *ATrous* .
- Non-mirrored *ATrous* algorithm, maybe as an option to *ATrousParametrize* where the maximum scale is not given by the data length, but such as coefficients do not need non-provided values (before and after the data chunk in time).

2.12.3 Class Cluster

Include file	**TeF_Cluster.hxx**
Constructors	`Cluster ()`
Public methods	`int Empty (void)`

```
int AddPix (const Pix& aPix)
int AddPix (const int aDim1, const int aDim2, const double aVal)
int SetPixLocalMax (const int aNum)

list<Pix>& GetAnchor (void)
int Absorb (list<Pix>& aListe)
void MoveCluster (const int aDim1, const int aDim2)
void SetCloseToBorder (void)
void SetImageNum (const int aNum)
void SetComposite (void)
void SetPhysical (const double aDim1Min, const double aDim1Step, const double aDim2Min, c

void SetPhysical(const Vector<double> aDim1V, const double aDim2Min, const double aDim2St

void SetPhysical(const Vector<double> aDim1V, const Vector<double> aDim1W, const double a

void ScaleEnergy (const double aScale)

Pix GetPix (const int aNum)
Point GetGeoBary (void)
Point GetWeightedBary (void)
Point GetPeakPoint (void)

int GetSize (void)
double GetWeight (void)
double GetMean (void)
double GetVariance (void)
double GetDim1Spread (void)
double GetDim2Spread (void)
double GetDim1Spread (double& aMin, double& aMax)
double GetDim2Spread (double& aMin, double& aMax)
double GetM11 (void)
double GetM02 (void)
double GetM20 (void)
double GetEqRadius (void)
int GetPeakNb (void)
double GetPeakPower (void)
bool GetCloseToBorder (void)
int GetImageNum (void)

void AppendFile (const string aFileName, const int aRefNum)

void SetDebug (const bool aValue)
bool GetDebug ()
```

The *cluster* class provides a list of *Pix* along with a list of methods for parameter extraction. Some methods are also used within the clustering algorithm, to create, expand and merge clusters as required.

The *Pix* structure holds two pixel (integer) coordinates *fAxis1, fAxis2* and a value *fVal* (double). It is used to store pixels of a cluster. Some operations on *Pix* have been implemented: *PixDistance* that returns the Euclidean distance, == that checks identity and < checking strict inferiority of both coordinates.

The *Point* structure holds two map (double) coordinates *fX1, fX2* and a value *fVal* (double). It is used to store the geometrical barycentre of a cluster. One operation is available, *PointDistance* , similar to *PixDistance* .

Two *AddPix* methods are provided, to allow for easy addition of pixels. Coordinates and values are needed, or a *Pix* object.

Merging of clusters requires the absorbed cluster to give a pointer to the absorbing cluster, using *GetAnchor* . The absorption is called by *Absorb* , that adds all elements from the *Pix* list of the absorbed cluster to the otherś before emptying the first list.

Element extraction is possible with *GetPix* to retrieve a *Pix* according to its entry order in the cluster (for lack of better discrimination) and *GetGeoBary* that returns the geometrical barycentre of the clusterś members.

Further parameter extraction is possible, and straightforward for the most part.
The *GetDim1Spread* and *GetDim2Spread* methods return the maximum coordinate difference between cluster members.
The *GetM11* , *GetM02* and *GetM20* methods provide geometrical moments of second order for the coordinates.

$$m_{11} = \frac{1}{N} \sum_{k=1}^{N} \mid (x_k - <x>) * (y_k - <y>) \mid$$

$$m_{20} = \frac{1}{N}\sum_{k=1}^{N}(x_k - < x >)^2$$

$$m_{02} = \frac{1}{N}\sum_{k=1}^{N}(y_k - < y >)^2$$

As for *GetEqRadius* , it estimates the radius (in number of pixels) of a cluster having a Gaussian energy profile (highest valued elements at the centre...) with

$$R = \frac{1.3 * \sigma_x}{< x >} + 1$$

so that a one pixel cluster returns 1.

Since clusters are built from map pixels, a transposition to physical coordinates, time and frequency, must be done at some point. That is the purpose of the *SetPhysical* methods.

2.12.4 Class Image

Include file **TeF_Image.hxx**

Constructors

```
Image (const int aDim1, const int aDim2,const bool aDebug)
Image (const Image &aIm)
```

Public methods

```
int Resize (const int aDim1, constintintImpaDrtm2(Matrix<double>& aMap)
int ReadMapFile (const string aFileName, Matrix<double>& aMap)

int Import (Matrix<double>& aMap, const Vector<double>& aProfile)
int Import (Matrix<double>& aMap, const Vector<double>& aProfile, const Vector<double>& a

int ImportAndNormalizeProgressive (Vector<double>& aIn, const double aThres1, const bool

int ImportAndMeanProgressive (Vector<double>& aIn, const double aThres1, const bool aCont

int ImportAndMeanSigmaProgressive (Vector<double>& aIn, const double aThres1, const bool

int ImportAndPercentileThresholdProgressive (Vector<double>& aIn, const double aPercent,


int BuildEnergyHisto (Matrix<double>& aMap)

int BuildFrequencyProfile (Matrix<double>& aMap, Vector<double>& aProfile)

int BuildFrequencyProfileAndSigma (Matrix<double>& aMap, Vector<double>& aProfile, Vector


int Thresholding (Matrix<double>& aMap, const double aThreshold)
int MeanThresholding (Matrix<double>& aMap)
int MeanAndSigmaThresholding (Matrix<double>& aMap, const Vector<double>& aMeans, const V
int MedianThresholding (Matrix<double>& aMap)
int SigmaThresholding (Matrix<double>& aMap, const double aTimesSigma)

int OtsuThresholding (Matrix<double>& aMap)
int LocalThresholding (Matrix<double>& aMap, const string aFileName)
int LocalThresholding (Matrix<double>& aMap, const Vector<double>& aVector)

int LocalThresholding (Matrix<double>& aMap)
int PercentileThresholding (Matrix<double>& aMap, const double aPercent)


int HKclustering (Matrix<double>& aMap)
int HKConnectedDim1Clustering (Matrix<double>& aMap)
int HKNoOWConnectedDim1Clustering (const Matrix<double>& aMap)
int HKConnectedDim1ProfileClustering (Matrix<double>& aMap, const Vector<double>& aProfil

int HKNoOWConnectedDim1ProfileClustering (const Matrix<double>& aMap, const Vector<double

int HKNoOWConnectedDim1ProfileClustering (const Matrix<double>& aMap, const Vector<double

int HKProgressiveClustering (Vector<double>& aVect)

int LocalMaxInfo (Matrix<double>& aMap)
int LocalMaxInfo (Matrix<double>& aMap, const int aRange)

int Connexion (const double aDilatation)
int ClusterCoincidence (void)
int ClusterCoincidence (const vector<Cluster>& aCV1, const double aScale1,const vector<Cl


void StackClusterEnergy (vector<Cluster>& aCV, const Matrix<double>& aMap, const int aInt

int CoherenceHK (const Matrix<double>& aMapA, const Matrix<double>& aMapB1, const Matrix<

int CoherenceHK (const Matrix<double>& aMapA, const vector<Matrix<double> >& aMapB1, cons

void SetOffsetBAdim2 (const int aVal)

int GetClusterNb (void)
int GetPixOnNb (void)

int GetLocalMask (Matrix<double>& aMap)
int GetClusterList(vector<Cluster>& aClusterList)
int GetLastClusterList(vector<Cluster>& aClusterList)

int FlushMapClusters (void)
int FlushMapConnections (void)

int WriteEnergyHisto (const string aFileName)
int WriteMapFile (Matrix<double>& aMap, const int aMaxDim1, const int aMaxDim, const str

int WriteClustersFile (const string aFileName)

void SetDebug (const bool aValue)
bool GetDebug ()
```

Some methods are not fully supported yet, and therefore best left alone. These are *Connexion*, *Cluster-Analysis* and *Tests*.

This class is supposed to provide image analysis tools, with a focus on time-frequency needs. It imports TF maps, allegedly built with *Transforms* to extract information. The end product is very restricted at the moment, a n-tuple file of cluster parameters, a fixed set of histograms.
Future development should probably give some leeway on the class output, and maybe offer some trigger method, unless a dedicated *Trigger* class is deemed necessary.

Nota bene: some analysis tools are sensitive to the axis signification, provided through the integer *aTransform*. Besides, other methods have only been implemented for a given orientation, indicated in their name.

At construction, the dimension of the map (a *Matrix* object) is asked, though it could be recovered. Internal objects are allocated memory in relation to these dimensions, which can be changed, without having to create a new object, via *Resize*.
Map import can be done from a *Matrix* with *Import* or an ASCII file using *ReadMapFile*.

It should be noted that the map is overwritten by several algorithms, most notably the clustering one. Setting the *aDebug* parameter to true makes most methods overwrite the map.

The first possible output is the histogram of the map pixel energy, built with *BuildEnergyHisto* and saved to a file by *WriteEnergyHisto*.
Note that some transforms are non-positive, thus making this histogram unreliable at the moment, since separation of negative and positive values is not implemented.
Are also available the mean and standard deviation profiles, computed with respect to time, as defined by the map orientation.

Map analysis starts with thresholding. Several methods have been implemented, without knowledge of their adequacy to our data type.

- *Thresholding* does hard thresholding at the given value (previous knowledge of map values is recommended).

- *MeanThresholding* cuts at the mean pixel value.
- *MeanAndSigmaThresholding* cuts at the given value, after normalizing both mean and standard deviation using the call values.
  Note that is normalization can be done using *Import*.
- *MedianThresholding* cuts at the pixel value median, that is mean between minimum and maximum values.
- *SigmaThresholding* cuts at the given fraction of the estimated sigma of the pixel values

$$\sigma =< E- < E >>$$

- *OtsuThresholding* uses the Otsu algorithm [27] to define the threshold. This relies on the hypothesis of a bimodal distribution of the pixel energy to isolate the higher values part.
- *LocalThresholding* methods are not fully supported, and try to estimate some zonal threshold.

- *PercentileThresholding* uses the pixel energy histogram, that must be constructed beforehand with *BuildEnergyHisto*.
  It then sums the histogram values till reaching the asked fraction of total pixel number. This method currently is meaningless on non-positive maps (see *BuildEnergyHisto*).

HKclustering [28] is in charge of the identification of pixel clusters, after determination of the pixel value threshold. It uses a strict definition of a cluster a group of connected pixels, with a 8-connectivity rule (horizontal, vertical and diagonal neighbors). It overwrites the map with zero for pixels below the threshold and a cluster number for those above.
Its main interest is filling a *list* of *Cluster* objects, thus achieving a first data reduction step.
Several implementations are proposed. They achieve cluster connection over map edges, or avoid map over-writing.

ClusterCoincidence tests the simultaneity of clusters, using strict conditions: the cluster with higher central frequency must have start and end times included within the time limits of the lower central frequency one.
Tentative methods for coherent analysis are included.
StackClusterEnergy adds to a given list of clusters the energy found at the corresponding pixels of the input map.
CoherenceHK stacks maps before applying the HK clustering algorithm. There is a reference map, and possibility of one or many delayed maps. For these, two consecutive maps are provided, in the hypothesis that while map size is the same for all channels, delays imply different time indexes will be requested.

GetClusterList returns the vector of extracted clusters. Note that this list can concern the map just processed or the previous one, if a connected clustering method was called. In the case, the latest map's clusters are accessed by *GetLastClusterList*.
To erase the clusters lists, use *FlushMapClusters*. To punctually suppress cluster map bridging, call *FlushMapConnections*.

2.12.5 Class Statistics

This class is in charge of both thresholding of cluster parameters and extraction of the values for the retained cluster and their copy to file. At the moment, the methods extracts a fixed set of parameters.

The parameters in use at the moment are the following:

- Cluster size.
- Cluster weight, the sum of pixel values in the cluster.
- Cluster position in dimension 1, estimated with the barycentre (see *Cluster*).
- Cluster position in dimension 2.
- Cluster moment 11.
- Cluster spread in dimension 1.
- Cluster spread in dimension 2.
- Cluster weight against cluster size (the weight is summed for a given size, to allow mean estimation, though without access to sigma).
- Cluster moment 11 against cluster size (similarly to above).
- Cluster spread in dimension 2 against position in dimension 1 (similarly to above).
  It should be noted that this particular output stands for a time spread distribution, and is therefore useful only when the map is correctly oriented.

Some more parameters seem interesting, and are to be implemented:

- Cluster size against dimension 1 and 2.
- Cluster energy against dimension 1 and 2.

Include file **TeF_Statistics.hxx**

Constructors

```
Statistics ()
Statistics (const bool aSaveOverUnderFlow, const bool aDebug)
```

Public methods

```
int SetDim1Set (const double aMin, const double aStep)
int SetDim1Set (const Vector<double> aV)

int SetDim1Weights (const Vector<double> aV)
int SetDim2Set (const double aMin, const double aStep)

int PowerTrigger (vector<Cluster>& aClusterList, const double aThreshold)

int PowerTrigger (list<Cluster>& aClusterMainList, vector<Cluster>& aClusterListIn, const

int PowerTriggerNLSpacing (list<Cluster>& aClusterMainList, vector<Cluster>& aClusterList

int PowerTrigger (vector<Cluster>& aClusterList, const double aThreshold, const string aF

int PowerAndPeakTrigger (list<Cluster>& aClusterMainList, vector<Cluster>& aClusterListIn

int PowerAndPeakTriggerNLSpacing (list<Cluster>& aClusterMainList, vector<Cluster>& aClus


int GetAllParameters (vector<Cluster>& aClusterList, const string aFileName)

int GetAllParameters (list<Cluster>& aClusterList, const string aFileName)

int GetSomeParameters (list<Cluster>& aClusterList, const string aFileName)

int GetStat(const string aFileName)

void SetDebug (const bool aValue)
bool GetDebug ()
```

The *DimXSet* methods inform of the physical values associated to the map indexes. They are used to call the *Cluster::ToPhysical* methods before applying the thresholds.

PowerTrigger only considers the total energy of a cluster, while *PowerAndPeakTrigger* also checks the maximum pixel energy of a cluster.

GetSomeParameters and —it GetAllParameters only defers by the list of parameters returned, but none gives the full available list.

2.13 BuC

to be written ...

2.14 BuLSet

This package is a logical package whose role is to manage and controle the installation of the BuL library. It also contains this document.

2.15 Appendix 1

One has defined the following enum:

- "BuS_Source.hxx"

```
typedef enum
{
  WhiteNoise,
  ColoredNoise
} Noise;
```

- "BuM_FFTW.hxx"

```
typedef enum          typedef enum
{                     {
  FFT_Complex,          FFT_Estimate,
  FFT_Real              FFT_Measure,
} Type;                 FFT_Patient,
        FFT_Exhaustive
                      } Wisdom;

typedef enum          typedef enum
{                     {
  Real,                 FFT_Forward,
  Imaginary,            FFT_Backward,
  Complex,              FFT_Both
  Modulus             } Direction;
} Output;
```

- "ITF_Interferometer.hxx"

```
typedef enum
{
  Virgo,
  Virgo_Quartz,
  Virgo_Quartz_Yag,
  Hanford_2k,
  Hanford_4k,
  Livingston,
  GEO,
  TAMA
} Detector;
```

- "Spectral_Taper.hxx"

```
typedef enum
{
  None,
  Hann,
  Welch,
  Haming,
  Bartlett
} Window;
```

- “Spectral_Spectral.hxx”

```
typedef enum
{
  One-Sided,
  Two-Sided
} PSDType;
```

2.16 Appendix 2

```
VIRGO

// 20 kHz sampling
static const int FZERO_VIRGO = 20000;

// VIRGO noise
// SIGMA_NOISE = SENS_NOISE * sqrt(FZERO/2)
static const double SIGMA_NOISE_VIRGO = 4.e-21;
static const double SENS_NOISE_VIRGO = 4.e-23;

// VIRGO expected sensitivity curv parameters
static const double SPEND_VIRGO = 1.2e-36;
static const double SMIRROR_VIRGO = 3.6e-43;
static const double SSHOT_VIRGO = 3.5e-46;
static const double FCUT_VIRGO = 500.;

static const double SPEND_QUARTZ = 4.5e-39;
static const double SMIRROR_QUARTZ = 3.6e-43;
static const double SSHOT_QUARTZ = 3.5e-46;
static const double FCUT_QUARTZ = 500.;

static const double SPEND_QUARTZ_YAG = 4.5e-39;
static const double SMIRROR_QUARTZ_YAG = 1.2e-44;
static const double SSHOT_QUARTZ_YAG = 3.5e-46;
static const double FCUT_QUARTZ_YAG = 500.;

// VIRGO geometry (angles are in degres)
static const double LONGITUDE_VIRGO = -10.5;
static const double LATITUDE_VIRGO = 43.63;
static const double AZIMUTHAL_VIRGO = 206.5;
static const double ARM_ANGLE_VIRGO = 90.0;

Hanford

// 16384 kHz sampling
static const int FZERO_HANFORD = 16384;

// Hanford noise
// SIGMA_NOISE = SENS_NOISE * sqrt(FZERO/2)
//static const double SIGMA_NOISE_HANFORD = 4.e-21;
//static const double SENS_NOISE_HANFORD = 4.e-23;

// Hanford_2k expected sensitivity curv parameters
static const double S1_2K = 1.94e40;
static const double S2_2K = 1.2e-36;
static const double S3_2K = 4.68e-46;
static const double S4_2K = 2.88e-46;
static const double FCUT_2K = 150.;

// Hanford_4k expected sensitivity curv parameters
static const double S1_4K = 1.94e40;
static const double S2_4K = 1.2e-36;
static const double S3_4K = 4.68e-46;
static const double S4_4K = 2.88e-46;
static const double FCUT_4K = 150.;

// Hanford geometry (angles are in degres)
static const double LONGITUDE_HANFORD = 119.41;
static const double LATITUDE_HANFORD = 46.45;
static const double AZIMUTHAL_HANFORD = 261.8;
static const double ARM_ANGLE_HANFORD = 90.0;
```

Livingston

```
// 16384 kHz sampling
static const int FZERO_LIVINGSTON = 16384;

// LIVINGSTON noise
// SIGMA_NOISE = SENS_NOISE * sqrt(FZERO/2)
//static const double SIGMA_NOISE_LIVINGSTON = 4.e-21;
//static const double SENS_NOISE_LIVINGSTON = 4.e-23;

// LIVINGSTON expected sensitivity curv parameters
static const double S1_LIVINGSTON = 1.94e40;
static const double S2_LIVINGSTON = 1.2e-36;
static const double S3_LIVINGSTON = 4.68e-46;
static const double S4_LIVINGSTON = 2.88e-46;
static const double FCUT_LIVINGSTON = 150.;


// LIVINGSTON geometry (angles are in degres)
static const double LONGITUDE_LIVINGSTON = 90.77;
static const double LATITUDE_LIVINGSTON = 30.56;
static const double AZIMUTHAL_LIVINGSTON = 333.0;
static const double ARM_ANGLE_LIVINGSTON = 90.0;
```

GEO

```
// 8192 kHz sampling
static const int FZERO_GEO = 8192;

// GEO noise
// SIGMA_NOISE = SENS_NOISE * sqrt(FZERO/2)
static const double SIGMA_NOISE_GEO = 4.e-21;
static const double SENS_NOISE_GEO = 4.e-23;

// GEO expected sensitivity curv parameters
static const double S1_GEO = 2.18e3;
static const double S2_GEO = 5.1e-43;
static const double S3_GEO = 2.0e-46;
static const double FCUT_GEO = 150.;
```

```
// GEO geometry (angles are in degres)
static const double LONGITUDE_GEO = - 9.81;
static const double LATITUDE_GEO = 52.25;
static const double AZIMUTHAL_GEO = 158.78;
static const double ARM_ANGLE_GEO = 94.33;


TAMA

// 8192 kHz sampling
static const int FZERO_TAMA = 8192;

// TAMA noise
// SIGMA_NOISE = SENS_NOISE * sqrt(FZERO/2)
//static const double SIGMA_NOISE_TAMA = 4.e-21;
//static const double SENS_NOISE_TAMA = 4.e-23;

// TAMA expected sensitivity curv parameters
static const double SPEND_TAMA = 7.68e-32;
static const double SMIRROR_TAMA = 5.2e-43;
static const double SSHOT_TAMA = 9.e-46;
static const double FCUT_TAMA = 400.;

// TAMA geometry (angles are in degres)
static const double LONGITUDE_TAMA = -139.54;
static const double LATITUDE_TAMA = 35.68;
static const double AZIMUTHAL_TAMA = 315.0;
static const double ARM_ANGLE_TAMA = 90.0;



AIGO

// AIGO geometry (angles are in degres)
static const double LONGITUDE_AIGO = -115.70;
static const double LATITUDE_AIGO = -31.35;
static const double AZIMUTHAL_AIGO = 90.; // unknown value
static const double ARM_ANGLE_AIGO = 90.0;
```

2.17 Appendix 3

a1b1g1	a2b1g1	a3b1g1	a4b1g1
a1b1g2	a2b1g2	a3b1g2	a4b1g2
a1b1g3	a2b1g3	a3b1g3	a4b1g3
a1b1g4	a2b1g4	a3b1g4	a4b1g4
a1b1g5	a2b1g5	a3b1g5	a4b1g5
a1b2g1	a2b2g1	a3b2g1	a4b2g1
a1b2g2	a2b2g2	a3b2g2	a4b2g2
a1b2g3	a2b2g3	a3b2g3	a4b2g3
a1b2g4	a2b2g4	a3b2g4	a4b2g4
a1b2g5	a2b2g5	a3b2g5	a4b2g5
a1b3g1	a2b3g1	a3b3g1	a4b3g1
a1b3g2	a2b3g2	a3b3g2	a4b3g2
a1b3g3	a2b3g3	a3b3g3	a4b3g3
a1b3g4	a2b3g4	a3b3g4	a4b3g4
a1b3g5	a2b3g5	a3b3g5	a4b3g5
	a2b4g2	a3b4g2	a4b4g2
	a2b4g3	a3b4g3	a4b4g3
	a2b4g4	a3b4g4	a4b4g4
	a2b4g5	a3b4g5	a4b4g5
	a2b5g4	a3b5g4	a4b5g4
	a2b5g5	a3b5g5	a4b5g5

Table 2.3: Table of the names of the 78 ZM waveforms

a1b1g1		a3b1g1	a4b1g1
			a4b1g2
a1b2g1		a3b2g1	
		a3b2g2	a4b2g2
			a4b2g3
		a3b2g4	
a1b3g1		a3b3g1	
a1b3g2		a3b3g2	
a1b3g3		a3b3g3	
a1b3g5		a3b3g5	
	a2b4g1		
		a3b4g2	
			a4b4g4
			a4b4g5
		a3b5g4	a4b5g4
			a4b5g5

Table 2.4: Table of the names of the 25 DFM waveforms

Bibliography

[1] CMT (v1r9) C. Arnault, Configuration Management Tool, http://www.lal.in2p3.fr/SI/CMT.htm

[2] Doxygen

[3] M. Frigo, S. G. Johnson, FFTW (v2r13) “The Fastest Fourier Transform in the West”, User’s Manual available at http://www.fftw.org

[4] http://tycho.usno.navy.mil/time.html
http://hpiers.obspm.fr/eop-pc

[5] D. B. Percival, A. T. Walden, “Spectral analysis for physical applications”, Cambridge University Press, 1993

[6] E. Cuoco, A. Viceré, “Mathematical Tools for Noise Analysis”, Virgo DAD, 1999

[7] P. D. Welch, IEEE Transactions on Audio and Electroacoustics, 15, 70-3, 1967

[8] E. Chassande-Mottin, *Testing the normality of the gravitational wave data with a low cost recursive estimate of the Kurtosis* , gr-qc/0212010

[9] A. V. Oppenheim, R. W. Schafer, *Discrete-time signal processing* , Prentice-Hall Signal Processing Series, New Jersay 1999

[10] L. R. Rabiner, B. Gold, *Theory and application of digital signal processing* , Prentice-Hall, New Jersay 1975

[11] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipies in Fortran*, Cambridge University Press 1992

[12] http://www.math.keio.ac.jp/ matumoto/emt.html

[13] C. W. Therrien, *Discrete Random Signals and Statistical Signal Processing*, Prentice-Hall, New Jersey 1992

[14] E. Cuoco, G. Curci, *Modelling a VIRGO like noise spectrum,* VIR-NOT-PIS-1390-095

[15] M Beccaria, E. Cuoco, G. Curci, *Adaptive Identification of VIRGO-like noise spectrum*, VIR-NOT-PIS-1390-096 and VIR-NOT-PIS-1390-095

[16] T. Damour, B. R. Iyer, B. Sathyaprakash *Phys. Rev.* D**63**, 044023 (2001)

[17] T. Zwerger, E. Mueller, *Astron. Astrophys.* **320**, 209 (1997)

[18] http://www.mpa-garching.mpg.de/Hydro/GRAV/index.html

[19] H. Dimmelmeier, J.A. Font, E. Mueller, *Astron. Astrophys.* **388**, 917 (2002)
H. Dimmelmeier, J.A. Font, E. Mueller, *Astron. Astrophys.* **393**, 523 (2002)

[20] http://www.mpa-garching.mpg.de/Hydro/RGRAV/index.html

[21] E. E. Flanagan, S. A. Hughes *Phys. Rev.* D**57**, 4535-4565 (1998)

[22] N. Arnaud *et al. Phys. Rev.* D**59**, 082002 (1999)

[23] T. Pradier *et al. Phys. Rev.* D**63**, 042002 (2001)

[24] N. Arnaud *et al. Phys. Rev.* D**67**, 062004 (2003)

[25] P. Flandrin, Temps fréquence, Hermes 1993

[26] P. Mallat, Exploration des signaux en ondelettes, Ellispes, 1999

[27] N. Otsu, “A Threshold Selection Method from Grey-Level Histograms”, IEEE Trans. Systems, Man, and Cybernetics **9-1**, 377 (1979)

[28] J. Hoshen, R. Kopelman, “Percolation and cluster distribution. I. Cluster multiple labelling technique and critical concentration algorithm” PRB**14**, 3438 (1976)

[29] M. Shensa, “The Discrete Wavelet Transform: Wedding the A Trous and Mallat Algorithms” IEEE Transactions on Signal Processing, **40-10**, 2464 (1992)

[30] I. Daubechies, W. Sweldens, “Factoring wavelet transforms into lifting steps”, J. Fourier Anal. Appl. **4-3**, 247 (1998)

[31] R. E. Kalman, “A new approach to linear filtering and prediction problems” ASME - J. Basic Eng.**82**, 35 (1960)

[32] R. G. Stockwell, L. Mansinha, R. P. Lowe, “Localization of the complex spectrum : the S transform”, IEEE Trans. Sig. Proc.**44**, 998 (1996)